TAMPEREEN TEKNILLINEN YLIOPISTO

TUOMAS VIITANEN

RÄÄTÄLÖITY .NET-KÄYTTÖLIITTYMÄ

Diplomityö

Tarkastaja: professori Ilkka Haikala
Tarkastaja ja aihe hyväksytty Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 7. lokakuuta 2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
VIITANEN, TUOMAS: Räätälöity .NET-käyttöliittymä
Diplomityö, 69 sivua
Kesäkuu 2010
Pääaine: ohjelmistotiede
Tarkastaja: professori Ilkka Haikala
Avainsanat: persoonallinen käyttöliittymä, periytyminen, .NET, Windows Forms, piirtäminen, välkkymätön piirtäminen, läpinäkyvyys, kohdistus, näppäimistön syöte, Windows Formsin sivuuttaminen

Räätälöidyllä käyttöliittymällä tarkoitetaan käyttöliittymää, jonka käyttäytyminen ja ulkoasu on toteutettu omista lähtökohdista periyttämistä apuna käyttäen. Tällöin ohjelmoija voi halutessaan toteuttaa käyttöliittymän jokaisen osan alusta asti vapaavalintaisella käyttäytymisellä ja ulkoasulla.

Vapaavalintaisuutta oli oltava eräässä projektissa, jossa toteutettiin mittalaitteen käyttöliittymä. Sitä oli tarkoitus voida käyttää tehdasolosuhteissa esimerkiksi paksuilla hansikkailla, joilla on hankala käyttää näppäimistöä ja hiirtä. Tämän takia valittiin kosketusnäyttö. Kyseisessä projektissa päädyttiin räätälöityyn käyttöliittymään, koska asiakasvaatimuksena käyttöliittymän piti olla havainnollinen eli sen piti mallintaa mittalaitetta realistisesti. Lisäksi kosketusnäytöstä ja käyttöolosuhteista seurasi erityisvaatimuksia käytettävyyden suunnittelulle ja siten käyttöliittymän olemukselle.

Diplomityön tavoitteena on selvittää, miten tehdään räätälöity käyttöliittymä Windowsiin, keskittyen erityisesti sen käyttäytymiseen ja ulkoasuun. Käyttäytymisestä selvitetään, miten räätälöity käyttöliittymä pääsee käsittelemään näppäimistön syötettä. Diplomityössä ei keskitytä hiiren syötteeseen. Ulkoasusta selvitetään, miten saadaan aikaan välkkymätön räätälöity käyttöliittymä ja miten läpinäkyvyys toimii. Lisäksi tavoitteena on tutkia sitä, miten mahdollisesti kohdatuista ongelmista tai rajoitteista päästään eroon.

Tavoitteet saavutettiin lähteitä tutkimalla. Lisäksi niiden puutteellisuuksien ja epäselvyyksien johdosta tehtiin pienimuotoisia kokeiluja asioiden selvittämistä tai varmistamista varten. Myös mittalaitteen käyttöliittymäprojektista saatiin merkittävää käytännön kokemusta.

Diplomityö antaa suosituksia konkreettisiksi toimenpiteiksi osin kokemukseen perustuen. Esimerkiksi liittyen tavoitteeseen tutkia rajoitteista eroon pääsemistä diplomityö esittelee mahdollisuuden sivuuttaa Windowsin valmiin arkkitehtuurin rajoitteet työläyden kustannuksella. Annetut suositukset ovat yleispäteviä, joskin harvinaisemmissa käyttötarkoituksissa voi olla perusteltua toimia toisin. Suosituksia voi käyttää Windowsin versiosta riippumatta, koska uudet versiot pohjautuvat vanhoihin lähinnä tuoden mukanaan uutta sisältöä vanhempiin verrattuna.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
VIITANEN, TUOMAS: Custom .NET User Interface
Master of Science Thesis, 69 pages
June 2010
Major: software science
Examiner: professor Ilkka Haikala
Keywords: personal user interface, inheritance, .NET, Windows Forms, painting, painting without flicker, transparency, focus, keyboard input, bypassing Windows Forms

Custom user interface is a user interface which behavior and appearance is implemented on custom basis by the means of inheritance. This allows the programmer to implement all parts of the user interface from scratch with arbitrary behavior and appearance.

Arbitrary behavior and appearance were needed in a project in which a user interface was implemented for an instrument. It was meant to be used in a factory environment meaning, for example, the usage of thick gloves that make the usage of keyboard and mouse difficult. Because of the environment, a touchscreen was selected. In the project a custom user interface was selected to be the implementation method to be used, because the client required the user interface to be illustrative. In other words, the user interface had to represent the instrument realistically. Also, the touchscreen and the environment produced special requirements for usability design and therefore also for the specification of the user interface.

The goal of this master of science thesis is to find out how a custom user interface is created for Windows. The main emphasis is on the behavior and appearance of the user interface. On the behavior's behalf, it is researched how a custom user interface can process keyboard input. The master of science thesis does not focus on mouse input. On the appearance's behalf, it is researched how a flicker-free custom user interface is created and how transparency works. Furthermore, a goal is to investigate how potentially encountered problems or restrictions are dealt with.

The goals were achieved by studying literature. Also, to overcome the shortcomings and ambiguities of the literature, some small scale experimental programs were made in order to sort out things or verify assumptions. Moreover, the actual implementation project of the user interface of the instrument offered significant practical experience.

The master of science thesis recommends some concrete actions based partly on experience. For example, related to the goal of investigating how restrictions are dealt with, the master of science thesis introduces an option for bypassing the restrictions of the general architecture of Windows at the expense of hard work. The given recommendations are universal, although it may be reasonable to disregard them in some rare situations. The recommendations can be used in spite of the version of Windows, because new versions are based on older ones, difference being mostly in the additional features provided by the new ones.

ALKUSANAT

Osallistuin diplomityön kirjoittamista varten suunnittelijana ja toteuttajana kahden vuoden ajan Atostekin projektiin, jossa mittalaitteelle tehtiin käyttöliittymä Windows-ympäristöön. Tämän jälkeen jatkoin aiheeseen perehtymistä kirjallisuuden ja pienimuotoisten kokeilujen avulla.

Diplomityön ohjaajana toimi Tuomas Fjällström, joka myös oli kyseisen projektin päällikkö. Hän antoi teknisiä ja arkkitehtuurillisia neuvoja projektin alkupuolella, mutta loppupuolella osaamiseni kehityttyä vastuuni laajeni samassa suhteessa. Projektissa työskenteli kulloinkin keskimäärin noin kolme henkilöä, joista kolmas vaihtui kolmeen kertaan.

Diplomityön esittämät oivallukset ovat peräisin kirjallisuudesta ja projektin henkilökunnalta. Kiitokset Tuomas Fjällströmin lisäksi kuuluvat Antti Tarvaiselle erityisesti ajatuksista liittyen hyvään lähteeseen, uuden tekniikan omaksumiseen sekä muistinhallintaan ja Jaakko Perkiölle erityisesti ajatuksista liittyen Windowsin viestien käsittelyyn käyttöliittymän osien käytön estämisen yhteydessä.

Kiitos Tuomas Fjällströmille hyvästä ja perinpohjaisesta ohjauksesta. Kiitos Ilkka Haikalalle kokeneen tarkastajan näkökulmasta. Kiitos Atostekille 350 kirjoitustunnin maksamisesta. Kiitos Juha Lemmetille ajatuksesta suunnitella aikataulu takaperin.

4.5.2010

Tuomas Viitanen
suuri@maililoota.cjb.net

SISÄLLYS

KÄSITTEIDEN MÄÄRITELMÄT

.NET-kehys	Tarkoittaa tässä diplomityössä .NET Framework versiota 2.0. .NET Framework on Microsoftin tekemä kehys, jonka avulla voidaan tehdä Windowsiin ohjelmia.
.NET-käyttöliittymä	Tarkoittaa tässä diplomityössä käyttöliittymää, joka on tehty .NET-kehyksen kokonaisuutta Windows Forms käyttäen.
ikkunakahva	Tarkoittaa kokonaislukua, jonka avulla Windows API -rajapinta tunnistaa kunkin kontrollin ymmärtämässään muodossa. Jokaiseen kontrolliin liittyy yksi ikkunakahva kontrollin alustamisen jälkeen. (windows handle)
ikkunaton kontrolli	Tarkoittaa itse tehtyä kontrollin korviketta, jolla on samankaltainen vastuualue kuin kontrollilla, mutta periytyy luokasta *System.Object* luokan *System.Windows.Forms.Control* sijaan. (windowless control, lightweight control)
ikkunattomien kontrollien säiliö	Tarkoittaa räätälöityä kontrollia, joka toimii säiliönä ikkunattomille kontrolleille.
kohdistus	Tarkoittaa valintaa siitä, mikä yksi käyttöliittymäikkuna ja mikä sen kontrollihierarkian kontrolleista on kulloinkin aktiivisena. Windows ohjaa näppäimistön syötteen aktiiviselle kontrollille, josta käytetään nimeä kohdistettu kontrolli. Kohdistus on olemassa, jotta Windows osaa ohjata näppäimistön syötteen käyttäjän valitsemaan kohteeseen. (focus)
kontrolli	Tarkoittaa luokkaa *System.Windows.Forms.Control* tai sen aliluokkaa. Käsitettä kontrolli käytetään kuvaamaan luokkaa tai siitä luotua oliota, mikä käy ilmi asiayhteydestä. (control)
kontrollien piirtojärjestys	Tarkoittaa sitä järjestystä, jossa kontrollihierarkian kontrollit piirretään.
kontrollihierarkia	Tarkoittaa kontrollien muodostamaa puumaista rakennetta, jossa on kontrolleja sisäkkäin ja jonka juurena on käyttöliittymäikkuna.
kontrollin piirtoalue	Tarkoittaa kontrollin aluetta, jolle kontrolli voi piirtää.

käyttöliittymä	Tarkoittaa kaikkia niitä teknisen järjestelmän piirteitä, joiden kautta ja avulla ihmisten on mahdollista käyttää järjestelmää sen tarkoitusta vastaavalla tavalla. Tässä diplomityössä keskitytään tietokoneiden käyttöliittymiin, jotka koostuvat näppäimistöstä, hiirestä ja näytöllä näkyvästä ohjelmasta.
käyttöliittymäikkuna	Tarkoittaa luokkaa *System.Windows.Forms.Form* tai sen aliluokkaa. (form)
mukautettu kontrolliolio	Tarkoittaa kontrollista luotua oliota, jonka ominaisuuksien arvoja on muutettu tai tapahtumia tilattu sen ulkopuolista käsittelyä varten.
mukautettu käyttöliittymä	Tarkoittaa käyttöliittymää, jonka käyttäytyminen ja ulkoasu on toteutettu omista lähtökohdista periyttämistä käyttämättä (ks. mukautettu kontrolliolio).
piirtopinta	Tarkoittaa rajapintaa, jolla voidaan piirtää erilaisille laitteille, kuten muisteille, näytönohjaimille, näytöille ja tulostimille.
räätälöity kontrolli	Tarkoittaa kontrollia, joka ei ole valmis kontrolli ja joka periytyy suoraan tai välillisesti valmiista kontrollista. *Form1* on esimerkki räätälöidystä kontrollista.
räätälöity käyttöliittymä	Tarkoittaa käyttöliittymää, jonka käyttäytyminen ja ulkoasu on toteutettu omista lähtökohdista periyttämistä apuna käyttäen (ks. räätälöity kontrolli).
säiliökontrolli	Tarkoittaa valmista kontrollia *ContainerControl* tai sen aliluokkaa. Esimerkiksi valmiit kontrollit *UserControl* ja *Form* ovat säiliökontrolleja.
valmis kontrolli	Tarkoittaa kontrollia, joka on nimiavaruudessa *System.Windows.Forms*.
Windows	Tarkoittaa tässä diplomityössä ilman tarkentavaa versiota versiota XP tai uudempaa.

Windows API -rajapinta	Tarkoittaa Windowsin matalan tason rajapintaa, joka mahdollistaa ohjelmoinnin Windowsiin. Windows API -rajapinta tarjoaa ohjelmallisen pääsyn esimerkiksi Windowsin ikkunointiin. Windows API -rajapinta on yksinään sekava kokoelma satoja sekalaisia C-ohjelmointikielellä kirjoitettuja palveluita. (myös Win32 API ja WinAPI)
Windows Forms	Tarkoittaa .NET-kehyksen kokonaisuutta, jonka avulla voidaan tehdä Windowsiin käyttöliittymä. (myös WinForms)
Windowsin viesti	Tarkoittaa tietueen *System.Windows.Forms.Message* esittämää viestiä. Windowsin viestit tulevat Windows API -rajapinnan läpi, ja niitä käytetään muun muassa näppäinpainalluksiin liittyvän tiedon välittämiseen. (Windows message)

1. JOHDANTO

Tässä diplomityössä käsitellään räätälöidyn käyttöliittymän tekemistä. Käyttöliittymäksi kutsutaan kaikkia niitä teknisen järjestelmän piirteitä, joiden kautta ja avulla ihmisten on mahdollista käyttää järjestelmää sen tarkoitusta vastaavalla tavalla [1]. Tässä diplomityössä keskitytään tietokoneiden käyttöliittymiin, jotka koostuvat näppäimistöstä, hiirestä ja näytöllä näkyvästä ohjelmasta.

Räätälöidyllä käyttöliittymällä tarkoitetaan käyttöliittymää, jonka käyttäytyminen ja ulkoasu on toteutettu omista lähtökohdista periyttämistä apuna käyttäen. Tällaisen käyttöliittymän osien käyttäytymisen ja ulkoasun määrittely on tarkkaa ja vapaata. Määrittely on tarkkaa, koska ohjelmoija näkee toteutuksen koodin ja voi muokata sitä. Määrittely on taas vapaata, koska halutessaan ohjelmoija voi toteuttaa jokaisen osan alusta asti vapaavalintaisella käyttäytymisellä ja ulkoasulla.

Tarkkuutta ja vapautta oli oltava eräässä projektissa, jossa toteutettiin mittalaitteen käyttöliittymä. Sitä oli tarkoitus voida käyttää tehdasolosuhteissa esimerkiksi paksuilla hansikkailla, joilla on hankala käyttää näppäimistöä ja hiirtä. Tämän takia valittiin kosketusnäyttö.

Mittalaitteen käyttöliittymä oli käyttöliittymä tässä diplomityössä ymmärretyssä merkityksessä, vaikkei kosketusnäytön yhteyteen asennettu näppäimistöä eikä hiirtä. Kosketusnäyttöön kuului nimittäin sisään rakennettu virtuaalinen näppäimistö, joka aukesi näkyville kosketusnäytön laidassa olevasta painikkeesta. Kosketusnäyttöön kuului myös sisään rakennettu virtuaalinen hiiri, jota ohjattiin painamalla kosketusnäyttöä sormella, kun virtuaalinen näppäimistö oli piilossa.

Käyttöliittymän oli tarkoitus mallintaa mittalaitteen rakenne. Mittalaitteessa oli mitattavien näytteiden ohjauksia eli näytelinjoja, jotka päätyivät erilaisia reittejä pitkin analysaattoriin. Näytelinja saattoi kulkea apulaitteiden läpi. Niinpä mittalaitteen rakenteen oli tarkoitus näkyä mittalaitteen eri osien tilaa esittävillä laitekuvilla ja virtaustietoja esittävillä putkistoilla. Mittalaitteen osien ja putkistojen muodostama rakenne vaihteli mittalaitteittain.

Käyttöliittymän oli tarkoitus näyttää myös erilaisia taulukoita. Niiden sisältö kuvasi tyypillisesti mittausjonoa, mittaustuloksia ja historiatietoa eri ajankohdilta. Mittausjono kuvasi sitä järjestystä, jossa näytelinjoja mitattiin.

Kosketusnäytön oli myös tarkoitus mahdollistaa mittalaitteen ohjauskomentojen antaminen. Esimerkkeinä ohjauskomennoista ovat näytelinjan jonkin mittauskerran prioriteetin nostaminen ja kokonaisen näytelinjan poistaminen tilapäisesti käytöstä.

Kyseisessä projektissa päädyttiin räätälöityyn käyttöliittymään, koska asiakasvaatimuksena käyttöliittymän piti olla havainnollinen eli sen piti mallintaa mittalaitetta rea-

listisesti. Lisäksi kosketusnäytöstä ja käyttöolosuhteista seurasi erityisvaatimuksia käytettävyyden suunnittelulle ja siten käyttöliittymän olemukselle.

Mittalaitteessa oli tietokone, jonka käyttöjärjestelmäksi valittiin Microsoftin Windowsin versio XP. Valintaan vaikutti se, että projektin alussa versiota Vista oltiin vasta julkaisemassa. XP oli tuolloin laajalti käytössä ja vakaa, kun taas Vista vaati tietokoneelta enemmän suoritustehoa. Sitä ei ollut paljon mittalaitteen tietokoneessa, joka oli suunniteltu tehdasolosuhteisiin.

Microsoftin silloinen suositus Windowsiin tehtäville uusille ohjelmille oli, että ne käyttävät Microsoftin tekemää kehystä nimeltään .NET Framework. Tämän kehyksen versiolla 2.0 kehitettiin ohjelmia XP:lle, joten tässä diplomityössä keskitytään .NET Framework versiota 2.0 käyttävän räätälöidyn käyttöliittymän tekemiseen.

.NET Framework 2.0 eroaa uudemmista versioista enimmäkseen siten, että uudemmissa on enemmän sisältöä. Tästä syystä diplomityössä esitellyt asiat soveltuvat pitkälti myös kyseisen kehyksen versioihin 3.0, 3.5 ja 4.0. Samoin, koska niillä kehitetään ohjelmia Windowsin versioille Vista ja 7, diplomityön esittelemät asiat eivät juurikaan menetä merkitystään uudempien käyttöjärjestelmien kohdalla.

Näin ollen, .NET-kehyksellä tarkoitetaan tässä diplomityössä vaihtoehtoa .NET Framework 2.0 ja Windowsilla ilman tarkentavaa versiota Microsoftin Windowsin versiota XP tai uudempaa. .NET-käyttöliittymällä puolestaan tarkoitetaan käyttöliittymää, joka on tehty .NET-kehyksen kokonaisuutta Windows Forms (myös WinForms) käyttäen. Sen avulla voidaan tehdä Windowsiin käyttöliittymä, ja tätä kokonaisuutta käsitellään tarkemmin kohdassa 2.1.

Diplomityössä ei tehdä rajauksia käytetyn ohjelmointikielen suhteen, koska .NET-kehykselle voidaan kirjoittaa ohjelmia eri ohjelmointikielillä. Terminologiana käytetään ohjelmointikielen C# terminologiaa, ja koodilistauksissa käytetään saman ohjelmointikielen versiota 2.0.

Diplomityön tavoitteena on selvittää, miten tehdään räätälöity .NET-käyttöliittymä, keskittyen erityisesti sen käyttäytymiseen ja ulkoasuun. Käyttäytymisestä selvitetään, miten räätälöity .NET-käyttöliittymä pääsee käsittelemään näppäimistön syötettä. Diplomityössä ei keskitytä hiiren syötteeseen. Ulkoasusta selvitetään, miten saadaan aikaan välkkymätön räätälöity .NET-käyttöliittymä ja miten läpinäkyvyys toimii. Lisäksi tavoitteena on tutkia sitä, miten mahdollisesti kohdatuista ongelmista tai rajoitteista päästään eroon.

Tavoitteet saavutettiin lähteitä tutkimalla. Lisäksi niiden puutteellisuuksien ja epäselvyyksien johdosta tehtiin pienimuotoisia kokeiluja asioiden selvittämistä tai varmistamista varten. Myös projektista, jossa mittalaitteelle tehtiin räätälöity .NET-käyttöliittymä, saatiin merkittävää käytännön kokemusta. Kirjoittaja osallistui kyseiseen projektiin suunnittelijana ja toteuttajana kahden vuoden ajan.

Diplomityön merkittävimmät lähteet ovat lähteet [2] ja [3]. Lähde [2] on .NET-kehyksen rajapinnan dokumentaatio, eikä kyseiseen lähteeseen viitata tekstissä enää myöhemmin. Lähde [3] puolestaan selittää, miten kyseistä rajapintaa käytetään. Kohtien pääasialliset lähteet on merkitty kaarisuluilla lukujen aluissa. Jokaisella kohdalla ei ole

pääasiallista lähdettä; sisältö perustuu tällöin kokemukseen ja pienimuotoisiin kokeiluihin.

Diplomityön sisältö jakautuu lukuihin seuraavasti. Aluksi luvussa 2 tarkastellaan .NET-käyttöliittymää yleisesti, esitellään käsitteet mukautettu kontrolliolio ja räätälöity kontrolli sekä verrataan niitä keskenään. Luvussa 3 käsitellään tarkemmin räätälöidyn kontrollin toteuttamista.

Luvussa 4 keskitytään räätälöidyn kontrollin piirtämiseen ja välkkymättömän .NET-käyttöliittymän tavoitteluun. Luvussa 5 taas esitellään kokonaisen kontrollihierarkian toimintaa yleisesti sekä kontrollihierarkiaa piirtämisen ja läpinäkyvyyden näkökulmasta. Luvussa 6 tutustutaan kohdistukseen ja näppäimistön syötteeseen.

Luvussa 7 esitellään mahdollisuus päästä eroon aiemmissa luvuissa kuvattujen toiminnallisuuksien rajoitteista työläyden kustannuksella. Lopuksi luvussa 8 on yhteenveto käsitellyistä asioista ja keskeisimmistä suosituksista sekä pohdinta diplomityön laajuudesta.

2. .NET-KÄYTTÖLIITTYMÄ

Tässä luvussa tarkastellaan .NET-käyttöliittymää yleisesti. Kohdassa 2.1 tutustutaan asioihin, joihin .NET-käyttöliittymä perustuu ([4; 3, s. 41–43]). Kohdassa 2.2 kuvataan .NET-käyttöliittymän rakennetta. Kohdassa 2.3 esitellään mukautettu käyttöliittymä vaihtoehtona räätälöidylle ([3, s. 9]), ja lopuksi näitä verrataan keskenään kohdassa 2.4.

2.1. Windows Forms

.NET-käyttöliittymää tehtäessä toteutetaan ohjelma, joka käyttää perustanaan .NET-kehyksen tarjoamia palveluita. Windows suorittaa .NET-kehystä käyttävät ohjelmat ajoaikaisessa ympäristössä nimeltään Common Language Runtime eli CLR.

.NET-kehys tarjoaa sitä perustanaan käyttäville ohjelmille rajapinnan, jossa on useita nimiavaruuksia. Tästä rajapinnasta käytetään nimeä .NET-rajapinta.

Yhtenä palveluna .NET-kehys tarjoaa kokonaisuuden Windows Forms, joka on pääosin .NET-rajapinnan nimiavaruudessa *System.Windows.Forms*. .NET-käyttöliittymällä tarkoitetaan juuri Windows Formsin avulla toteutettua ja sitä perustanaan käyttävää käyttöliittymää. Windows Forms on siis teknologia, joka mahdollistaa CLR:n vuorovaikutuksen .NET-käyttöliittymän kanssa: CLR tulkitsee .NET-käyttöliittymän Windowsin matalan tason rajapintaan. Tämä rajapinta on nimeltään Windows API (myös Win32 API ja WinAPI), ja se mahdollistaa ohjelmoinnin Windowsiin. Windows API -rajapinta tarjoaa ohjelmallisen pääsyn esimerkiksi Windowsin ikkunointiin.

Windows API -rajapinta on yksinään sekava kokoelma satoja sekalaisia C-ohjelmointikielellä kirjoitettuja palveluita. Windows Forms käyttää siis sisäisesti Windows API -rajapintaa tarjoten sitä helpomman tavan käyttöliittymien ohjelmointiin Windowsiin.

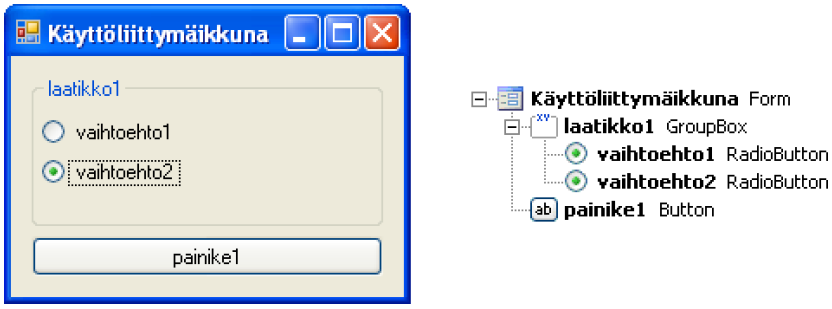
Toisin sanoen, ohjelmoija käyttää .NET-käyttöliittymässä Windows Formsin tarjoamia luokkia. CLR tunnistaa nämä luokat ja käsittelee Windowsin matalan tason yksityiskohdat kutsumalla Windows API -rajapintaa.

2.2. .NET-käyttöliittymän rakenne

.NET-käyttöliittymä näkyy yhden tai useamman käyttöliittymäikkunan (form) kautta. Käyttöliittymäikkunalla tarkoitetaan luokkaa *System.Windows.Forms.Form* tai sen aliluokkaa. Käyttöliittymäikkunan sisällä on kontrolleja (control). Ne edustavat .NET-käyttöliittymän osia, jotka näkyvät käyttäjälle. Kontrolli tarkoittaa luokkaa

System.Windows.Forms.Control tai sen aliluokkaa, joten käyttöliittymäikkunakin on kontrolli.

Kontrollin sisällä voi olla muita kontrolleja, mutta kukin kontrolli voi olla kerrallaan enintään vain yhden toisen kontrollin sisällä. Kontrollit muodostavat näin puumaisen rakenteen, jossa on kontrolleja sisäkkäin ja jonka juurena on käyttöliittymäikkuna. Tästä rakenteesta käytetään nimeä kontrollihierarkia. Kontrollihierarkiassa ei voi olla kehärakennetta, jossa kontrollin sisällä on toinen kontrolli, jonka sisällä itse samaan aikaan on. Esimerkiksi kuvan 2.1 käyttöliittymäikkunan sisällä on kontrollit *laatikko1* ja *painike1*. Kontrollin *laatikko1* sisällä taas on kontrollit *vaihtoehto1* ja *vaihtoehto2*. Kuvan oikeassa laidassa on vastaava kontrollihierarkia.



Kuva 2.1. *Käyttöliittymäikkuna ja vastaava kontrollihierarkia (mukailtu lähteestä [3, s. 10])*

Jos kontrollin sisällä on muita kontrolleja, ne ovat tallessa sen ominaisuudessa (property) *Controls*, joka on kokoelma kontrolleja. Kontrollin ominaisuudessa *Parent* on puolestaan tallessa kontrollin sisältävä kontrolli.

2.3. Mukautettu kontrolliolio ja räätälöity kontrolli

Mukautetulla käyttöliittymällä tarkoitetaan käyttöliittymää, jonka käyttäytyminen ja ulkoasu on toteutettu omista lähtökohdista periyttämistä käyttämättä. Mukautettu kontrolliolio on siis kontrollista luotu olio, jonka ominaisuuksien arvoja on muutettu tai tapahtumia tilattu (subscribe) sen ulkopuolista käsittelyä varten. Näin voidaan, riippuen kontrollin mukauttamista varten tarjoaman rajapinnan kattavuudesta, muuttaa joiltain osin yksittäisen olion käyttäytymistä ja ulkoasua.

Kuvassa 2.2 näkyy, miten valmiin kontrollin *Button* olioita on mukautettu näkymään erinäköisenä vain niiden ominaisuuksien arvoja muuttamalla. Valmiilla kontrollilla tarkoitetaan kontrollia, joka on nimiavaruudessa *System.Windows.Forms*. Myös *painike1* on mukautettu, koska siihen on asetettu esimerkiksi teksti.

Kontrollin ominaisuuksilla mukautetaan myös asetteluun liittyvää tietoa. Kontrollin ominaisuudella *Size* määritetään kontrollin koko. Kontrollin ominaisuudella *Location* taas määritetään kontrollin sijainti sisältävän kontrollin asettelussa. Kuvassa valmiin kontrollin *Button* oliot on aseteltu näiden kahden ominaisuuden avulla valkoiseen sisältävään kontrolliin.

Kuva 2.2. *Valmiin kontrollin Button viisi mukautettua kontrollioliota*

Kuten aiemmin mainittiin, räätälöity käyttöliittymä toteutetaan periyttämistä käyttäen. Räätälöity kontrolli taas on siis kontrolli, joka on periytetty valmiista kontrollista. Tarkemmin sanottuna räätälöity kontrolli on kontrolli, joka ei ole valmis kontrolli ja joka periytyy suoraan tai välillisesti valmiista kontrollista.

2.4. Räätälöidyn kontrollin edut

Räätälöidyllä kontrollilla on useita etuja mukautettuihin kontrolliolioihin verrattuna. Räätälöidystä kontrollista voidaan esimerkiksi luoda monta samanlaista oliota siinä missä mukauttaminen on oliokohtaista. Räätälöity kontrolli paketoi siis uuden käyttäytymisen ja ulkoasun uudelleenkäytettävään luokkaan. Jos räätälöidyn kontrollin tilalla käytettäisiin mukautettuja kontrolliolioita, mukauttava koodi jouduttaisiin toistamaan jokaisen mukautetun kontrolliolion yhteydessä.

Jokaisen räätälöidystä kontrollista tehdyn olion ei kuitenkaan pidä olla samanlainen. Räätälöity kontrolli tarjoaa oliokohtaista mukauttamista varten saman rajapinnan kuin kantaluokkansakin. Lisäksi räätälöity kontrolli voi laajentaa tätä rajapintaa.

Räätälöidyn kontrollin etuna on myös laajempi muunneltavuus, koska valmiiden kontrollien aliluokilleen tarjoama rajapinta on kattavampi kuin mukauttamista varten tarjoama. Näin on, koska valmiin kontrollin ulkopuolisesti käytettävissä olevat ominaisuudet ja tapahtumat ovat samalla aliluokankin käytettävissä. Tämän lisäksi aliluokka pääsee käyttämään kantaluokkansa suojattuja (protected) jäseniä (member) sekä voi ylikirjoittaa monia rakentajia, ominaisuuksia ja metodeja, mikä puolestaan ei ole mukauttamisessa mahdollista. Valmiiden kontrollien tapahtumia ei tyypillisesti voi ylikirjoittaa, mutta niitä nostavat metodit voi.

Mukauttamisen rajat voidaan siis ohittaa käyttämällä räätälöityä kontrollia. Siinä missä kontrollin käyttäytymisen ja ulkoasun muuttamiseen käytetään mukauttamisessa kontrollin ominaisuuksia ja tapahtumia, räätälöity kontrolli puolestaan ylikirjoittaa valmiin kontrollin jäseniä samaa tarkoitusta varten. Tästä seuraa, että mikä tahansa mukautettu kontrolliolio voidaan paketoida räätälöidyksi kontrolliksi siten, että se asettaa rakentajassaan vastaavat ominaisuudet ja käsittelee tapahtumansa sisäisesti. Tällainen rää-

tälöity kontrolli voi ylikirjoittaa kantaluokkansa jokaisen rakentajan kutsumaan jotain metodia, joka asettaa kyseiset ominaisuudet.

Jäseniään ylikirjoittamalla räätälöity kontrolli pääsee mukautetun kontrolliolion paketoimisen lisäksi vaikuttamaan siihen, miten Windows Forms vuorovaikuttaa räätälöidyn kontrollin kanssa. Windows Forms asettaa käyttöliittymäikkunan ja muiden kontrollien ominaisuuksia sekä kutsuu kontrollien metodeja. Windows Forms tyypillisesti esimerkiksi välittää näin kontrollille näppäimistön syötettä tai ohjeistaa kontrollia piirtämään itsensä. Nämä ominaisuudet ja metodit ovat ylikirjoitettavia, eli vuorovaikutusta ei ole piilotettu räätälöidyltä kontrollilta. [3, s. 9–10.] Räätälöity kontrolli voi siis muuttaa käyttäytymistä ja ulkoasua perinpohjaisesti.

Toisentyyppisenä etuna räätälöidyllä kontrollilla on mahdollisuus lisätä rajapintaansa paremmin jotain tiettyä käyttötarkoitusta palvelevia jäseniä. Tällöin kyseinen käyttö helpottuu, koska lisätty rajapinta kapseloi epäoleellisia yksityiskohtia pois tai mahdollistaa halutun muotoisen tiedon käytön. Esimerkiksi tiedostojärjestelmän hakemistoja selaava ohjelma on todennäköisesti helpompi toteuttaa räätälöidyllä kontrollilla *HakemistoTreeView* kuin valmiilla kontrollilla *TreeView*, joka ei sellaisenaan tarjoa hakemiston käsitettä. [3, s. 321.]

3. RÄÄTÄLÖIDYN KONTROLLIN TOTEUTTAMINEN

Tässä luvussa käsitellään räätälöidyn kontrollin toteuttamista. Kohdassa 3.1 otetaan kantaa räätälöidyn kontrollin kantaluokan valintaan ([3, s. 321–323]). Kohdassa 3.2 taas tutustutaan siihen, miten räätälöity kontrolli seuraa kantaluokkansa tapahtumaa tai omaa tapahtumaansa. Lopuksi kohdassa 3.3 esitellään lyhyesti, miten räätälöity kontrolli pääsee vaikuttamaan käyttäytymiseensä ja ulkoasuunsa ([5]). Viimeinen kohta toimii johdantona luvuille 4–6.

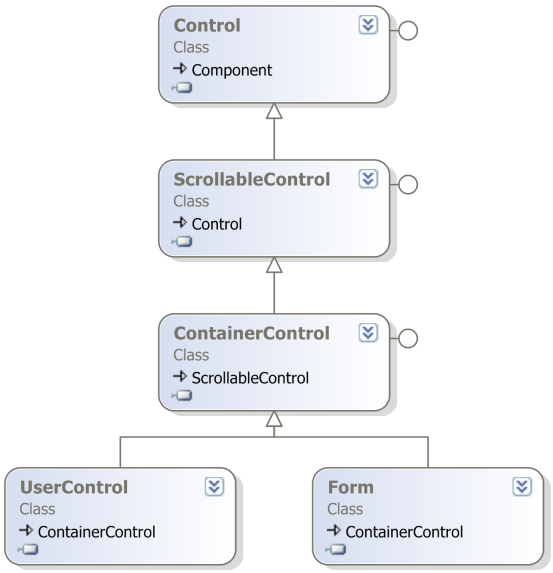
3.1. Kantaluokan valinta

Valmiit kontrollit, kuten monet muutkin .NET-rajapinnan luokat, on suunniteltu periytettäviksi, mikä helpottaa räätälöityjen kontrollien toteuttamista. Tällaisen suunnittelun ansiosta myös valmiiden kontrollien käyttäytymisen ja ulkoasun perinpohjainen muuttaminen on mahdollista räätälöidyissä kontrolleissa, kuten kohdassa 2.4 todettiin.

Koska valmiiden kontrollien koko luokkahierarkia tukee periyttämistä, räätälöidyn kontrollin kantaluokaksi kannattaa valita se valmis kontrolli, jonka valmiina tarjoamista palveluista räätälöity kontrolli hyötyy eniten. Toisin sanoen muutostarvetta kannattaa minimoida. Räätälöity kontrolli saa kaiken valmiissa kontrollissa olevan käyttäytymisen ja ulkoasun periytymisen ansiosta valmiina, mistä voi olla haittaakin: kantaluokka voi joskus liikaa rajata räätälöidyn kontrollin mahdollisuuksia muokata käyttäytymistään ja ulkoasuaan kohtuullisella lisätyöllä.

Vähiten räätälöidyn kontrollin mahdollisuuksia rajaavat valmiiden kontrollien luokkahierarkian ylimmät luokat. Nämä luokat ovat nimeltään *Control*, *ScrollableControl*, *ContainerControl*, *UserControl* ja *Form*. Niiden luokkahierarkia näkyy kuvassa 3.1.

Kuvan valmiista kontrolleista *Control* tarjoaa siitä periytetylle räätälöidylle kontrollille kaikki yleiset palvelut, kuten perusedellytykset toimia ja näkyä osana .NET-käyttöliittymää. Perusedellytyksiin kuuluu esimerkiksi tuki näppäimistön ja hiiren syötteen vastaanottamiseen. Valmis kontrolli *ScrollableControl* puolestaan lisää yleisiin palveluihin tuen vieritykselle. Valmis kontrolli *ContainerControl* taas lisää tuen sisällä olevien kontrollien kohdistuksen hallinnointiin. Kohdistusta käsitellään tarkemmin kohdissa 6.1–6.5 ja kyseistä tukea kohdissa 6.3 ja 6.10.



Kuva 3.1. *Valmiiden kontrollien luokkahierarkian ylimmät luokat (mukailtu lähteestä [3, s. 324])*

Seuraavina ovat valmiit kontrollit *UserControl* ja *Form*. Valmis kontrolli *UserControl* lisää alustamista varten tapahtuman *Load* ja tarjoaa ohjelmankehitysympäristölle, esimerkiksi Visual Studio 2005:lle, tuen, jonka ansiosta ohjelmoija voi asetella ja muokata kyseisen valmiin kontrollin sisälle tulevia kontrolleja suunnitteluaikana (design-time). Suunnitteluajalla tarkoitetaan ohjelmankehitysympäristön tilaa, jossa voidaan muokata visuaalisesti .NET-käyttöliittymää. Valmis kontrolli *Form* lisää taas palvelut, joiden avulla siitä periytetty räätälöity kontrolli voi näkyä käyttöliittymäikkunana .NET-käyttöliittymässä.

Valmis kontrolli *UserControl* soveltuu esimerkiksi muita kontrolleja kokoavaksi säiliöksi yhdistäen ne yhdeksi loogiseksi kokonaisuudeksi. Sellainen voi olla esimerkiksi tekstikenttien kokoelma, joka näkyy käyttöliittymäikkunoissa toistuvasti samanlaisena. [3, s. 321–322; 6.]

Jos räätälöidyn kontrollin sisälle ei tule kontrolleja, valmiit kontrollit *UserControl* ja *ContainerControl* eivät tarjoa aliluokkanaan olevalle räätälöidylle kontrollille mitään tarpeellista, toisin kuin kantaluokkansa. Tällaisen räätälöidyn kontrollin kantaluokka kannattaa valita valmiiden kontrollien *ScrollableControl* ja *Control* väliltä. [6; 7.] Näin vältetään kantaluokan ylimääräisiä palveluita ja noudatetaan yleistä periaatetta, että aliluokka on oikeasti samaa tyyppiä kuin kantaluokka. Periaate tarkoittaa tässä yhteydessä sitä, että valmiin kontrollin *ContainerControl* jokaisen aliluokan on tarkoitus olla muita kontrolleja sisältävä kontrolli.

Mitä tahansa valmiista kontrolleista *Control*, *ScrollableControl*, *ContainerControl*, *UserControl* ja *Form* voi siis käyttää räätälöidyn kontrollin kantaluokkana. Näille yhteisenä asiana räätälöidyn kontrollin vastuulle jää käyttäytymisensä ja ulkoasunsa toteuttaminen suurilta osin alusta asti. Toteuttamistyö helpottuu huomattavasti, jos räätälöidylle kontrollille löytyy sopiva kantaluokka muualta valmiiden kontrollien luokkahierarkiaa kuin ylimmistä luokista. Toteuttamistyö helpottuu, koska luokkahierarkiassa alempana

olevat luokat on erikoistettu johonkin tiettyyn tarkoitukseen, ja ne tarjoavat siten enemmän palveluja valmiina.

Riippumatta siitä, harkitaanko kantaluokaksi valmiiden kontrollien luokkahierarkian ylimpiä tai muita luokkia, yleisenä kantaluokan valintaohjeena on, että kannattaa valita luokkahierarkiassa mahdollisimman matalalla oleva luokka, jonka tyyppinen aliluokka oikeasti vielä on ja joka ei rajaa liikaa aliluokan mahdollisuuksia käyttötarkoitustaan varten. Mitä ylempää luokkahierarkiasta valinta tehdään, sitä enemmän menetetään valmiita palveluita kustannuksena lisääntyneelle vapaudelle. Kannattaa kiinnittää huomiota siihen, millaisia palveluita mikäkin vaihtoehto tarjoaa ja mitä näistä palveluista räätälöity kontrolli voi käyttää.

3.2. Tapahtuman seuraaminen nostavalla metodilla

Kuten kohdassa 2.4 mainittiin, Windows Forms vuorovaikuttaa kontrollien kanssa kutsumalla niiden ylikirjoitettavia metodeja. Ne voivat olla suojattuja, mutta Windows Forms voi silti käyttää niitä. Jotkin vuorovaikutuksessa käytettävistä metodeista ovat sellaisia, jotka nostavat kontrollin jonkin tapahtuman. Tällöin .NET-rajapinnassa nostavan metodin nimi eroaa nostavaan metodiin liittyvän tapahtuman nimestä ainoastaan etuliitteen osalta. Esimerkiksi nostava metodi *OnPaint* liittyy tapahtumaan, jonka nimi on *Paint*. Vastaavaa nimeämiskäytäntöä on suositeltavaa käyttää myös räätälöidyissä kontrolleissa luettavuuden takia.

Räätälöity kontrolli voi seurata kantaluokkansa tapahtumaa tai omaa tapahtumaansa kahdella tavalla. Räätälöity kontrolli voi tilata kyseisen tapahtuman tai käyttää vastaavaa nostavaa metodia. Jälkimmäisessä vaihtoehdossa räätälöity kontrolli ylikirjoittaa kantaluokkansa nostavan metodin tai muokkaa oman nostavan metodinsa toteutusta.

Jotta kantaluokkansa nostavan metodin ylikirjoittava räätälöity kontrolli välittää tapahtuman eteenpäin, ylikirjoittavassa metodissa on muistettava kutsua kantaluokan toteutusta. Siellä tapahtuu kantaluokan käsittely tapahtumalle ja edelleen sen nostaminen. Samoin omaa nostavaa metodia muokkaavassa räätälöidyssä kontrollissa on kyseisessä metodissa muistettava nostaa itse tapahtuma. Räätälöidyn kontrollin ei pidä välittää tapahtumaa eteenpäin, jos räätälöity kontrolli käsitteli tapahtuman siten, ettei halua käsittelyn enää jatkuvan.

.NET-rajapinnassa suositellaan, ettei räätälöity kontrolli seuraa kantaluokkansa tapahtumaa tai omaa tapahtumaansa tilaamalla kyseisen tapahtuman. Tämä on ymmärrettävää, koska suositusta noudatettaessa räätälöity kontrolli on tunnetusti ensimmäinen tai viimeinen, joka pääsee käsittelemään itseensä liittyvää tapahtumaa. Tapahtuman tilannut räätälöity kontrolli olisi sen sijaan käsittelyjärjestyksessä tuntemattomalla paikalla.

Räätälöity kontrolli voi käsitellä itseensä liittyvää tapahtumaa ennen sen eteenpäin välittämistä tai vasta sen jälkeen. Tavallisesti on toivottua, että räätälöity kontrolli pääsee käsittelemään itseensä liittyvää tapahtumaa ensimmäisenä, jolloin räätälöity kontrolli ensin käsittelee kyseistä tapahtumaa ja vasta sitten välittää sen eteenpäin. Joissain ti-

lanteissa voi olla kuitenkin perusteltua, että järjestys on juuri päinvastainen tai räätälöity kontrolli suorittaa käsittelyä sekä ennen että jälkeen.

3.3. Käyttäytyminen ja ulkoasu

Valmiilla kontrollilla on useita jäseniä, jotka räätälöity kontrolli voi ylikirjoittaa toiminnallisuuden lisäämistä varten. Näiden avulla räätälöidylle kontrollille voidaan toteuttaa tarkoituksenmukainen käyttäytyminen ja ulkoasu. Seuraavaksi esitellään valmiin kontrollin muutamia metodeja, joita Windows Forms kutsuu ja joiden avulla räätälöity kontrolli pääsee vaikuttamaan käyttäytymiseensä tai ulkoasuunsa. Tällaisen vaikuttamisen yksityiskohtiin keskitytään myöhemmin muissa luvuissa.

Räätälöity kontrolli piirtää itsensä ylikirjoittamalla taustan piirtoon tarkoitetun metodin *OnPaintBackground* ja sisällön piirtoon tarkoitetun metodin *OnPaint*. Metodia *OnPaint* vastaava tapahtuma on siis sellainen, jota räätälöity kontrolli seuraa ylikirjoittamalla kyseisen metodin. Metodi *OnPaintBackground* on poikkeus, koska kyseinen metodi ei nosta mitään tapahtumaa nimestään huolimatta. Räätälöidyn kontrollin piirtämistä esitellään tarkemmin luvussa 4. Piirtämistä näiden kahden metodin avulla esitellään tarkemmin kohdassa 4.1. Kontrollihierarkia vaikuttaa siihen, miten piirtäminen toimii, mitä taas tarkastellaan luvussa 5.

Räätälöity kontrolli vastaanottaa näppäimistön syötettä ylikirjoittamalla metodit *OnKeyDown*, *OnKeyPress* ja *OnKeyUp*. Nämä metodit nostavat tapahtumia, joita räätälöity kontrolli seuraa samoin ylikirjoittamalla tapahtumaan liittyvän metodin. Näppäimistön syötteen vastaanottamista esitellään tarkemmin luvussa 6, ja kohdassa 6.13 esitellään tarkemmin näihin kolmeen metodiin liittyviä tapahtumia.

4. RÄÄTÄLÖIDYN KONTROLLIN ULKOASU

Tässä luvussa tarkastellaan räätälöidyn kontrollin ulkoasun toteuttamista. Kohdassa 4.1 tutustutaan räätälöidyn kontrollin piirtämiseen metodien *OnPaintBackground* ja *OnPaint* avulla. Kohdissa 4.2–4.3 esitellään sitä, miten räätälöity kontrolli voi ottaa vastuun kantaluokan ulkoasun elementtien piirtämisestä tai siirtää vastuun itse valitsemalleen oliolle (kohdassa 4.3 [3, s. 389–403, s. 514–520]).

Kohdassa 4.4 keskitytään piirtokertaan ([3, s. 214–218]), jonka aloittamiseen taas perehdytään kohdassa 4.5 ([3, s. 214–218]). Seuraavaksi kohdassa 4.6 esitellään piirtopinta, jolle piirtäminen tapahtuu ([3, s. 211–214, s. 219–221]). Lopuksi kohdissa 4.7–4.10 tarkastellaan räätälöidyn kontrollin piirtämisen optimointia välkkymättömän .NET-käyttöliittymän tavoittelua varten (kohdassa 4.8 [3, s. 249–252, s. 414, s. 867], kohdassa 4.9 [3, s. 244–249, s. 403–404, s. 839], kohdassa 4.10 [3, s. 839–843]).

4.1. Piirtäminen metodeilla *OnPaintBackground* ja *OnPaint*

Räätälöity kontrolli toteuttaa ulkoasunsa eli piirtää itsensä metodin *OnPaintBackground* ylikirjoittamisen ja tapahtuman *Paint* seuraamisen avulla. Piirtäminen tapahtuu samoilla metodeilla riippumatta siitä, toteuttaako räätälöity kontrolli ulkoasunsa alusta asti vai muokkaako räätälöity kontrolli kantaluokaltaan saamaansa ulkoasua.

Jos räätälöity kontrolli toteuttaa ulkoasunsa alusta asti, räätälöity kontrolli piirtää kaiken itse. Tällöin räätälöity kontrolli ei kutsu kantaluokkaansa toteutettua piirtämistä ylikirjoitetuista metodeistaan *OnPaintBackground* ja *OnPaint*.

Kantaluokan metodin *OnPaint* kutsumatta jättäminen aiheuttaa sen, ettei räätälöity kontrolli nosta tapahtumaa *Paint*. Tämä ei haittaa, koska räätälöity kontrolli piirsi jo itsensä eikä piirtämistä ole tässä yhteydessä tarkoitettu tapahtuvan kantaluokassa eikä missään ulkopuolisessa metodissa, jolle tapahtuma *Paint* on tilattu. Yleisestikin, kantaluokan piirtoa ei pidä kutsua, ellei halua kantaluokan piirtämää sisältöä näkyville [3, s. 408, s. 519].

Jos räätälöity kontrolli taas muokkaa kantaluokaltaan saamaansa ulkoasua, räätälöity kontrolli piirtää kantaluokkaansa toteutetun piirtämisen jälkeen. Tällöin räätälöity kontrolli piirtää kantaluokkansa piirtämän sisällön päälle siten, että kantaluokan piirtämästä sisällöstä osa jää yhä näkyviin. Räätälöity kontrolli siis kutsuu ensin kantaluokkansa metodia *OnPaintBackground* tai *OnPaint* ja vasta sitten alkaa piirtää itse.

4.2. Kantaluokan elementtien piirtäminen

Joihinkin valmiisiin kontrolleihin on suunniteltu mekanismi, jonka ansiosta valmis kontrolli voi kutsua metodiensa *OnPaintBackground* ja *OnPaint* piirtokoodista silloin tällöin piirtokoodia, jonka räätälöity kontrolli toteuttaa. Tällainen mekanismi mahdollistaa sen, että räätälöity kontrolli voi halutessaan ottaa vastuun valmiin kontrollin ulkoasun elementtien piirtämisestä. Ulkoasun elementeillä tarkoitetaan käyttöliittymän osan ulkoasun osia, joihin ulkoasu voidaan mielekkäästi jakaa.

Jotkin valmiit kontrollit tarjoavat vastaavan mekanismin toisessa muodossa. Tällöin valmiin kontrollin metodit *OnPaintBackground* ja *OnPaint* kutsuvat valmiille kontrollille asetetun olion piirtokoodia. Piirtämisestä vastaavan olion tyyppi ei ole kontrolli, ja valmiin kontrollin piirtämisestä vastaava olio voidaan vaihtaa ajoaikana. Räätälöity kontrolli voi vapaasti valita käyttämänsä olion, ja tavallisesti räätälöidyn kontrollin toteuttamisen yhteydessä periytetään piirtämistä varten luokka valmiina tarjottavien vaihtoehtojen joukosta.

Valmis kontrolli tarjoaa molempien mekanismien yhteydessä eksplisiittisesti rajapinnan, jonka avulla räätälöity kontrolli voi siirtää itselleen kantaluokkansa vastuun piirtää ulkoasunsa elementtejä. Ensimmäisessä mekanismissa rajapinta perustuu valmiin kontrollin tapahtumiin siinä missä jälkimmäisessä taas valmiin kontrollin ominaisuuteen. Jälkimmäinen mekanismi kattaa tavallisesti kantaluokan ulkoasun jokaisen elementin piirtämisen toisin kuin ensimmäinen.

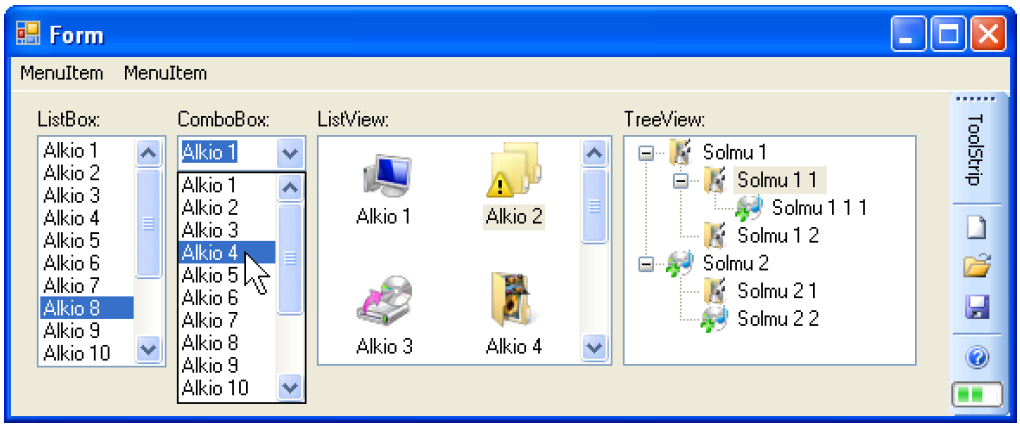
Jos valmis kontrolli tarjoaa rajapinnan, jolla ulkoasun elementit voidaan piirtää muualla, rajapinta samalla tarjoaa valmiin kontrollin sisäisestä tilasta kaiken riittävän tiedon piirtämistä varten. Jos tällaisella rajapinnalla tapahtuvaa piirtämistä verrataan aiemmin kohdassa 4.1 esiteltyyn piirtämiseen, joka tapahtuu osin valmiin kontrollin piirtämän sisällön päälle, jälkimmäisen tavan puutteena on tavallisesti se, ettei valmis kontrolli tarjoa piirtämistä varten riittävästi tietoa helposti käytettävässä muodossa. Lisäksi tällöin voi hyvinkin olla, että ainoa tapa selvittää valmiin kontrollin sisäinen tila on tutkia valmiin kontrollin piirtämää sisältöä ohjelmallisesti.

Räätälöidyn kontrollin ei kannata tutkia kantaluokkansa piirtämää sisältöä ohjelmallisesti työläyden takia, ellei tutkittava asia ole triviaali. Tällöinkin piirretyn sisällön tutkiminen on kyseenalaista, koska menetelmä on virhealtis johtuen siitä, että kaikkia mahdollisia kantaluokan piirtämiä asioita voi olla vaikea ennakoida. Lisäksi kantaluokan uuden version mukanaan tuomat ulkoasun muutokset voivat rikkoa räätälöidyn kontrollin päättelyn.

Jos valmiin kontrollin ulkoasun elementit halutaan piirtää toisin, mutta valmis kontrolli ei tarjoa tähän rajapintaa eikä sopivasti tietoa sisäisestä tilastaan, varmin ratkaisu on toteuttaa vastaava räätälöity kontrolli alusta asti luokasta *Control* periyttämällä. Tämä on työlästä verrattuna mahdollisuuteen piirtää ulkoasun elementit rajapinnalla, joka tarjoaa riittävät tiedot. Moni valmis kontrolli tarjoaakin kyseisen mahdollisuuden, koska valmiiden kontrollien ulkoasujen elementtejä halutaan tyypillisesti piirtää toisin. Kohdassa 4.3 tarkastellaan esimerkkejä tällaisista valmiista kontrolleista.

4.3. Esimerkkejä piirrettävistä elementeistä

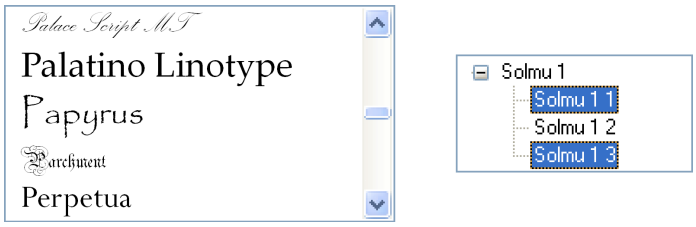
Jos valmis kontrolli tarjoaa ulkoasunsa elementtien piirtämiseen rajapinnan, joka perustuu tapahtumiin, valmiilla kontrollilla on ominaisuus *OwnerDraw* tai *DrawMode*. Tämän ominaisuuden avulla räätälöity kontrolli määrittää ulkoasun ne elementit, joiden piirtämisestä räätälöity kontrolli ottaa vastuun. Jos räätälöity kontrolli ottaa tällaista vastuuta, räätälöidyn kontrollin on seurattava joitakin kantaluokkansa tapahtumia ja toteuttaa niiden yhteyteen piirtokoodi. Esimerkiksi valmiilla kontrolleilla *MenuItem*, *ListBox*, *ComboBox*, *ListView* ja *TreeView* on ominaisuus *OwnerDraw* tai *DrawMode*. Kyseiset valmiit kontrollit on esitetty kuvassa 4.1.



Kuva 4.1. *Valmiit kontrollit MenuItem, ListBox, ComboBox, ListView, TreeView ja ToolStrip*

Jos valmis kontrolli tarjoaa ulkoasunsa elementtien piirtämiseen rajapinnan, joka perustuu taas ominaisuuteen ja sen arvon edellyttämään, piirtämisestä vastaavan luokan tyyppiin, valmiilla kontrollilla on ominaisuus *Renderer*. Räätälöity kontrolli asettaa tähän ominaisuuteen olion, joka toteuttaa ulkoasun jokaisen elementin piirron. Esimerkiksi valmiilla kontrollilla *ToolStrip* on ominaisuus *Renderer*. Tämäkin valmis kontrolli näkyy kuvassa.

Seuraavaksi käsitellään tarkemmin valmiiden kontrollien *ListBox*, *ListView*, *TreeView* ja *ToolStrip* ulkoasujen elementtien piirtämistä. Näistä valmis kontrolli *ListBox* kuvaa luetteloa, *ListView* luettelonäkymää, *TreeView* puunäkymää ja *ToolStrip* valikkopalkkia. Kuvan 4.2 vasemmassa laidassa on esimerkiksi luettelo, jonka alkiot räätälöity kontrolli piirtää kantaluokkansa sijaan käyttäen eri kirjasinlajeja. Saman kuvan oikeassa laidassa on taas puunäkymä, johon räätälöity kontrolli on lisännyt puuttuvan monivalintamahdollisuuden. Sen lisääminen kuvataan lähteessä [3, s. 396–403]. Lisääminen on mahdollista, koska räätälöity kontrolli voi piirtää puunäkymän solmut.



Kuva 4.2. *Räätälöity kontrolli piirtää kantaluokan alkiot eri kirjasinlajeilla (mukailtu lähteestä [3, s. 395]) ja toinen räätälöity kontrolli toteuttaa monivalinta-mahdollisuuden (mukailtu lähteestä [3, s. 402])*

Valmis kontrolli *ListBox* nostaa tapahtumiaan *MeasureItem* ja *DrawItem* luettelon alkioiden piirtämistä varten. Tapahtuma *MeasureItem* nostetaan, jos luettelon alkioiden keskinäinen korkeus vaihtelee. Tämän tapahtuman avulla räätälöity kontrolli ilmoittaa valmiille kontrollille, minkä kokoisena aikoo kunkin alkion piirtää. Räätälöity kontrolli ilmoittaa asiasta asettamalla tapahtuman *MeasureItem* parametrin ominaisuudet *ItemHeight* ja *ItemWidth*.

Tapahtuma *DrawItem* taas nostetaan, kun luettelon alkio pitää piirtää. Tapahtuman *DrawItem* parametri tarjoaa valmiin kontrollin *ListBox* sisäisestä tilasta kaiken piirrolle riittävän tiedon, kuten piirrettävän alkion tilan. Se selviää kyseisen parametrin ominaisuudesta *State*.

Valmis kontrolli *ListView* taas nostaa tapahtumiaan *DrawColumnHeader*, *DrawItem* ja *DrawSubItem* luettelonäkymän otsikoiden ja alkioiden piirtämistä varten. Valmis kontrolli *TreeView* puolestaan nostaa tapahtumaansa *DrawNode* puunäkymän solmujen piirtämistä varten.

Valmiiden kontrollien ulkoasujen elementtien piirtämistä varten nostettujen tapahtumien parametrit voivat sisältää ominaisuuden *DrawDefault*. Sen avulla räätälöity kontrolli voi valita ulkoasun elementin piirron tapahtuvan sittenkin valmiissa kontrollissa. Lisäksi tapahtumien parametrit voivat sisältää metodeja, joiden nimi alkaa sanalla draw. Näiden avulla räätälöity kontrolli voi suorittaa joitakin vaiheita, jotka sisältyvät valmiin kontrollin suorittamaan piirtoon.

Valmis kontrolli *ToolStrip* ei nosta tapahtumiaan ulkoasunsa elementtien piirtämistä varten, vaan kutsuu ominaisuuteensa *Renderer* asetetun olion metodeja. Tämä olio on tyypiltään *ToolStripRenderer*, josta on valmiiksi periytetty kaksi aliluokkaa. Mistä tahansa näistä kolmesta periytettäessä vain ne metodit on ylikirjoitettava, joita haluaa muuttaa. Valinta kolmen luokan välillä kannattaakin tehdä siten, että muutoksia on tehtävä mahdollisimman vähän. Pelkkien värimuutosten yhteydessä riittää valita valmis luokka *ToolStripProfessionalRenderer* ja periyttää sen rakentajalle parametri luokasta *ProfessionalColorTable*.

Luokan *System.Windows.Forms.ToolStripManager* staattinen rajapinta tarjoaa erilaisia palveluita valmiista kontrollista *ToolStrip* luoduille olioille. Aiheeseen liittyen, rajapinta mahdollistaa saman piirtämisestä vastaavan olion asettamisen kerralla jokaiselle valmiista kontrollista *ToolStrip* luodulle oliolle, jonka ominaisuus *RenderMode* on oletusarvossaan.

4.4. Piirtokerta

Kontrolli piirtää elinkaarensa aikana useita kertoja. Kontrolli piirtää esimerkiksi aina, kun sen näytettävä sisältö muuttuu tai jokin päällä oleva kontrolli siirtyy sivuun paljastaen enemmän alla olevaa kontrollia. Tällöin sen on piirrettävä piirtoalueensa juuri esiin tullut uusi alue. Kontrollin piirtoalue tarkoittaa aluetta, jolle kontrolli voi piirtää, ja piirtoaluetta käsitellään tarkemmin kohdassa 5.7.

Piirtoalueelle edellisellä kerralla piirretty jää pohjaksi seuraavalle piirtokerralle. Jos seuraavalla piirtokerralla kontrolli ei siis piirräkään koko piirtoaluettaan, piirtämättä jääneisiin osiin jää näkyviin aiemman piirtokerran jälkeensä jättämä sisältö. Tämä voi olla peräisin myös jostain toisesta kontrollista, joka voi jopa sisältyä toiseen käyttöliittymäikkunaan. Toisin sanoen, aiemmin samalla näytön kohdalla näkyneen toisen käyttöliittymäikkunan piirtämä alue on siis saatava päivittymään uuden käyttöliittymäikkunan kyseisellä kohdalla näkyvän kontrollin mukaiseksi.

Koska piirtoalueelle jää edellisen piirtokerran sisältö pohjaksi, kontrollin piirtokerta toimii siten, että Windows pyytää kontrollia ensin pyyhkimään taustansa esiin. Vasta tämän jälkeen Windows pyytää kontrollia piirtämään varsinaisen sisältönsä.

Kun kontrolli pyyhkii taustansa esiin, kontrolli piirtää koko piirtoalueellensa pelkän taustan. Näin kontrolli peittää aluksi kaiken aiemmin samalle näytön kohdalle piirretyn sisällön.

Kun kontrolli piirtää varsinaisen sisältönsä, kontrollin ei ole pakko piirtää koko piirtoaluettaan. Vaikka osia kontrollin piirtoalueesta jäisikin tällöin piirtämättä, kontrolli piirsi näihin osiin jo silloin, kun kontrolli pyyhki taustansa esiin. Näin kontrollin piirtokerran päätyttyä kontrollin koko piirtoalue tulee piirrettyä, ja varsinaisen sisällön piirron ei ole keskityttävä taustan esiin pyyhkimiseen.

Kun Windows pyytää kontrollia pyyhkimään taustansa esiin, Windows Forms kutsuu kontrollin metodia *OnPaintBackground*. Kun taas Windows pyytää kontrollia piirtämään sisältönsä, vastaavasti Windows Forms kutsuu kontrollin metodia *OnPaint*.

Tavallisesti räätälöidyn kontrollin metodi *OnPaintBackground* piirtää esimerkiksi harmaalla värillä täytetyn, koko piirtoalueen levyisen ja korkuisen laatikon ja räätälöidyn kontrollin metodi *OnPaint* puolestaan piirtää vain osalle piirtoaluetta. Jos kuitenkin tämä metodi piirtää koko piirtoalueelle, jäljelle ei jää mitään taustaa, jonka esiin pyyhkimisestä olisi hyötyä. Tällöin räätälöidyn kontrollin metodi *OnPaintBackground* voidaan toteuttaa tyhjäksi [3, s. 244].

4.5. Piirtokerran aloittaminen

Kontrollin piirtokerran voi aloittaa Windows tai ohjelmaan kirjoitetut suorat metodikutsut. Samat metodit päätyvät kutsuttaviksi myös Windowsin tapauksessa. Molemmissa tapauksissa ensin merkitään kontrollin piirtoalue osittain tai kokonaan epäkelvoksi ja sitten kontrollia ohjeistetaan piirtämään itsensä.

Kontrollin piirtoalueen osa merkitään epäkelvoksi kontrollin metodilla *Invalidate*. Useita kontrollin piirtoalueen osia merkitään epäkelvoiksi kyseistä metodia useasti käyttämällä. Samalla metodilla merkitään myös koko piirtoalue epäkelvoksi.

Kontrollia ohjeistetaan piirtämään itsensä metodillaan *Update*. Tämän metodin kutsumisen jälkeen Windows Forms kutsuu kyseiselle kontrollille ensin metodia *OnPaintBackground* ja sitten metodia *OnPaint*.

Kun kontrollia ohjeistetaan piirtämään itsensä, kontrolli voi piirtää ainoastaan piirtoalueensa niille osille, jotka ovat epäkelpoja. Kontrollin ei pidä voida piirtää muille osille, koska niiden sisältö on yhä ajan tasalla. Piirtoalueen epäkelpojen alueiden ansiosta piirtäminen voi tapahtua tehokkaammin ja piirron välkkyminen voi vähentyä (tähän palataan tarkemmin kohdassa 4.8). Kun kontrollia, jonka piirtoalueella ei ole ainuttakaan epäkelpoa osaa, ohjeistetaan piirtämään itsensä, kontrolli ei voi piirtää mitään.

Räätälöidyn kontrollin on merkittävä piirtoalueensa osia epäkelvoiksi sitä mukaan, kun räätälöidyn kontrollin sisäinen tietosisältö muuttuu ja vaikuttaa räätälöidyn kontrollin ulkoasuun eli siihen, millä tavoin räätälöidyn kontrollin on tarkoitus piirtää itsensä. Muutoin räätälöidyn kontrollin piirtämä sisältö ei pysy ajan tasalla räätälöidyn kontrollin sisäisen tietosisällön kanssa.

Windows Forms ohjeistaa aika ajoin kontrolleja, joiden piirtoalueessa on epäkelpo osa, piirtämään itsensä. Tämän takia räätälöidyn kontrollin ei tavallisesti ole ohjeistettava itseään näin heti sen jälkeen, kun on merkinnyt piirtoalueensa osan epäkelvoksi. Tällainen suunnitteluratkaisu mahdollistaa tehokkaamman ja vähemmän välkkymistä sisältävän piirron, koska kontrollin yksi piirtokerta voi kattaa kerralla monta osaa, jotka ovat kontrollin piirtoalueessa epäkelpoja.

Joskus räätälöidyn kontrollin on kuitenkin ohjeistettava itseään piirtämään itsensä. Näin on tyypillisesti silloin, kun räätälöity kontrolli esittää edistymistä aikaa vievälle laskennalle, joka tapahtuu samassa säikeessä kuin räätälöidyn kontrollin koodi suoritetaan. Tällöin Windows Forms ei pääse laskennan väliin ohjeistamaan räätälöityä kontrollia piirtämään itseään, jolloin räätälöidyn kontrollin sisäinen tietosisältö ajautuu väkisin kauas siitä, miten räätälöity kontrolli pääsi piirtämään itsensä edellisellä kerralla. Ratkaisuna ongelmaan räätälöity kontrolli kutsuu metodinsa *Invalidate* jälkeen metodiaan *Update*. Näin räätälöidyn kontrollin ulkoasu pysyy ajan tasalla.

Räätälöidyn kontrollin metodia *Update* ei pidä kutsua ilman syytä, koska kutsuminen pakottaa mahdollisesti rasituksen alla olevan Windowsin piirtämään heti, vaikka olisi olemassa joitain kriittisempiäkin tehtäviä. Jos tällöin kutsutaankin ainoastaan kontrollin metodia *Invalidate*, Windows voi valita hetken, jolloin siihen mennessä kertyneet, epäkelpoja piirtoalueen osia sisältävät kontrollit piirretään.

Räätälöidyn kontrollin piirtokertaa ei pidä aloittaa kutsumalla räätälöidyn kontrollin metodia *OnPaint* parametrilla, joka on luotu räätälöidyn kontrollin metodilla *CreateGraphics*, koska tällöin luovutaan Windows Formsin tarjoamasta arkkitehtuurillisesta ratkaisusta piirtää kontrolleja. Oman uuden arkkitehtuurin keksiminen ei ole tarpeellista, koska Windows Forms tarjoaa jo riittävän ratkaisun. On pienempi työ tutustua

valmiiseen arkkitehtuuriin kuin suunnitella vastaava alusta asti ja ottaa oma arkkitehtuuri käyttöön valmiin päälle.

4.6. Piirtopinnalle piirtäminen

Piirtopinta on rajapinta, jolla voidaan piirtää erilaisille laitteille, kuten muisteille, näytönohjaimille, näytöille ja tulostimille. Kun räätälöity kontrolli piirtää piirtoalueelleen metodeissaan *OnPaintBackground* tai *OnPaint* tai jokin osapuoli piirtää valmiin kontrollin ulkoasun elementtejä (kohta 4.2), piirtäminen tapahtuu nimenomaan piirtopinnalle.

.NET-kehys tarjoaa GDI+-rajapinnan (Graphics Device Interface), joka on suurilta osin .NET-rajapinnan nimiavaruudessa *System.Drawing*. Kyseisen nimiavaruuden luokka *Graphics* kapseloi GDI+-rajapinnan mukaisen piirtopinnan.

GDI+-rajapinta on tilaton, minkä takia luokan *Graphics* piirtäville metodeille on aina määritettävä koordinaatit. GDI+-rajapinnan käyttöä esitellään enemmän lähteessä [3, s. 211–252].

Räätälöity kontrolli siis piirtää piirtoalueelleen kutsumalla tyyppiä *Graphics* olevan olion metodeja. Tällöin räätälöity kontrolli piirtää piirtopinnalle. Räätälöity kontrolli saa piirtämiseen tarkoitettuun metodiinsa tyyppiä *Graphics* olevan olion metodin parametrin välityksellä. Esimerkiksi metodi *OnPaint* saa tyyppiä *PaintEventArgs* olevan parametrin, jonka ominaisuutta *Graphics* käyttämällä kyseinen metodi pääsee piirtämään piirtopinnalle.

Piirtopinnalle piirretään esimerkiksi tekstiä, geometrisia kuvioita ja kuvia. Geometristen kuvioiden reunat piirretään tavallisesti luokan *System.Drawing.Pen* avulla. Geometristen kuvioiden sisäpuoli taas täytetään tavallisesti luokan *System.Drawing.Brush* avulla. Luokan *Graphics* rajapintaa esitellään enemmän lähteessä [3, s. 219–220].

4.7. Piirtämisen yleinen optimointi

Piirtämisen optimointi tehostaa piirtämistä, jolloin piirron välkkyminen vähenee. Välkkymätön .NET-käyttöliittymä näyttää ammattimaiselta ja erottuu muista [3, s. 244, s. 867].

Piirtokoodia, kuten mitä tahansa koodia, voidaan optimoida yleisten ohjelmistotekniikan suunnitteluperiaatteiden avulla. Seuraavaksi käydään läpi kaksi esimerkkiä. Ensin esitellään ylimääräisen laskennan välttäminen ja sen jälkeen uusien resurssien varaamisen viivästäminen.

Ylimääräistä laskentaa päätyy tavallisesti piirtokoodiin esimerkiksi siten, että piirtokoodin alussa uusi resurssi varataan ja lopussa vapautetaan, vaikka resurssi varataan aina samanlaisena eri piirtokerroilla. Resurssin varaaminen ja vapauttaminen vievät ylimääräistä aikaa, joten on tehokkaampaa varata sama resurssi vain kerran esimerkiksi räätälöidyn kontrollin rakentajassa. Jos resurssi saattaa muuttua joskus, mutta useimmiten resurssi säilyy piirtokertojen välillä samana, resurssi voidaan silti laittaa talteen rää-

tälöidyn kontrollin yksityiseen kenttään. Piirtokoodissa pitää välttää esimerkiksi uusien olioiden luomista luokista *Pen* ja *Brush* [3, s. 849].

Resurssin varaaminen voidaan myös viivästää siihen hetkeen, jolloin resurssia oikeasti käytetään. Tällöin on säilytettävä tallessa resurssin varaamisessa käytettävät tiedot, jotta resurssi voidaan varata milloin vain. Jos nämä tiedot muuttuvat räätälöidyn kontrollin elinkaaren aikana, muutokset voidaan ottaa huomioon siten, että tietoihin liittyvä resurssi merkitään vanhentuneeksi. Jos vanhentunutta resurssia yritetään käyttää, vanhentunut resurssi ensin vapautetaan, sitten varataan uusi korvaava resurssi ja lopuksi uusi resurssi annetaan käytettäväksi. Näin on tehty listauksessa 4.1.

Listauksen räätälöity kontrolli *AnaloginenKello* on optimoitu varaamaan yksityisessä kentässään *viisarinReuna* olevan resurssin vasta käytön yhteydessä. Ratkaisun etuna jokainen muutos ominaisuuksiin *ViisarinReunanVäri* ja *ViisarinReunanLeveys* ei edellytä vanhan resurssin vapauttamista ja uuden varaamista.

Uusien resurssien varaamisen viivästämisestä on hyötyä sitä enemmän, mitä enemmän räätälöidyllä kontrollilla on ominaisuuksia, jotka määrittävät resurssin varaamisessa käytettävät tiedot. Kyseinen optimointi tekee tehokkaammaksi esimerkiksi koodin, joka aika ajoin asettaa räätälöidyn kontrollin kaikki ominaisuudet. Lisäoptimointina räätälöity kontrolli voi merkitä resurssin vanhentuneeksi vain, jos resurssiin vaikuttavaan ominaisuuteen asetetaan arvo, joka poikkeaa edellisestä arvosta.

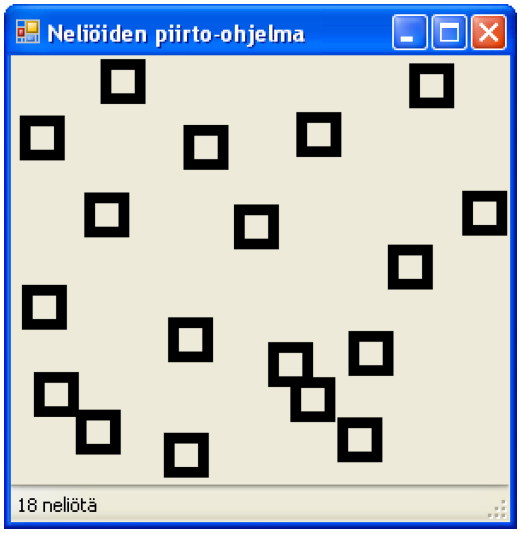
Kohdissa 4.8–4.10 keskitytään piirtämisen optimointiin tavoilla, jotka eivät niinkään liity yleisiin ohjelmistotekniikan suunnitteluperiaatteisiin. Esiteltävillä tavoilla voidaan vähentää piirron välkkymistä huomattavasti.

```
namespace Viitanen.Diplomityö
{
    using System.Drawing;
    using System.Windows.Forms;

    public class AnaloginenKello : Control
    {
        private Color viisarinReunanVäri = Color.Sienna;
        public Color ViisarinReunanVäri
        {
            get { return viisarinReunanVäri; }
            set
            {
                viisarinReunanVäri = value;
                viisarinReunaOnVanhentunut = true;
                Invalidate(); //uusi väri on saatava näkyviin
            }
        }

        private float viisarinReunanLeveys = 1f;
        public float ViisarinReunanLeveys
        {
            // Koodi sivuutettu, vastaava kuin yllä.
        }

        private bool viisarinReunaOnVanhentunut = true;

        // Resurssi, jota käytetään ominaisuuden kautta.
        private Pen viisarinReuna;
        private Pen ViisarinReuna
        {
            get
            {
                if ( viisarinReunaOnVanhentunut )
                {
                    if ( viisarinReuna != null )
                    {
                        // Vapauta vanha resurssi.
                        viisarinReuna.Dispose();
                    }

                    // Varaa uusi resurssi.
                    viisarinReuna = new Pen( viisarinReunanVäri,
                                             viisarinReunanLeveys );
                    viisarinReunaOnVanhentunut = false;
                }

                return viisarinReuna;
            }
        }

        // Muu koodi sivuutettu.
    }
}
```

Listaus 4.1. *Resurssin varaaminen viivästetysti (mukailtu lähteestä [3, s. 409–410])*

4.8. Piirtämisen optimointi rajaamalla

Räätälöidyn kontrollin piirtoa optimoidaan rajaamalla siten, että räätälöity kontrolli merkitsee vain piirtoalueensa muuttuneet osat epäkelvoiksi koko piirtoalueensa sijaan. Esimerkkinä tästä on kuvassa 4.3 oleva neliönpiirto-ohjelma, joka lisää neliön aina, kun käyttäjä painaa jotain kohtaa hiirellä. Neliöt eivät ole kontrolleja, ja käyttöliittymäikkuna piirtää neliöt silmukassa.



Kuva 4.3. *Neliönpiirto-ohjelma (mukailtu lähteestä [3, s. 249])*

Jos neliönpiirto-ohjelma toimii siten, että jokaisen hiiren painalluksen seurauksena koko käyttöliittymäikkunan piirtoalue merkitään epäkelvoksi, jokaisen uuden neliön lisääminen aiheuttaa sen, että seuraava piirtokerta piirtää kaikki vanhatkin neliöt. Kun neliöitä on satoja, vanhat neliöt välkkyvät tällöin selvästi jokaisen hiiren painalluksen yhteydessä.

Jos neliönpiirto-ohjelma taas toimiikin siten, että käyttöliittymäikkunan metodia *Invalidate* käytetään oikein, jokaisen hiiren painalluksen yhteydessä merkitään vain se käyttöliittymäikkunan piirtoalueen osa epäkelvoksi, johon uusi neliö lisätään. Tällöin käyttöliittymäikkunan metodin *OnPaint* parametrinaan saama, tyyppiä *PaintEventArgs* oleva olio sisältää ominaisuuden *ClipRectangle* ja se kattaa piirtoalueen kaikki epäkelvot osat. Koska piirtokoodi ei vaikuta piirtopinnan siihen osaan, joka on ominaisuuden *ClipRectangle* kuvaaman osan ulkopuolella, piirron tehokkuus paranee ja piirron välkkyminen vähenee radikaalisti.

Vaikka metodia *Invalidate* käytetään oikein, metodin *OnPaint* koko piirtokoodi yhä suoritetaan. Tällöin ominaisuuden *ClipRectangle* kuvaaman osan ulkopuolelle ulottuva piirto menee hukkaan. Lisäoptimointina metodi *OnPaint* voikin piirtää ainoastaan ne neliöt, jotka ovat osin tai kokonaan ominaisuuden *ClipRectangle* kuvaaman osan sisäpuolella. Tällöin metodi *OnPaint* tarkastaa neliöt piirtävässä silmukassaan jokaisen neliön sijainnin ennen neliön piirtämistä ja jättää jotkin neliöt piirtämättä. Tällaisesta lisäoptimoinnista on hyötyä lähinnä, jos näin vältetään aikaa paljon vievää piirtokoodia.

Räätälöity kontrolli voi hyödyntää ominaisuutta *ClipRectangle* myös metodissaan *OnPaintBackground*, jonka yhteydessä kyseinen ominaisuus toimii samoin kuin metodin *OnPaint* kanssa. Metodin *OnPaintBackground* on siis pyyhittävä tausta esiin ainoastaan ominaisuuden *ClipRectangle* kuvaaman osan sisäpuolelta. Tällaisesta optimoinnista on käytännössä hyötyä esimerkiksi silloin, kun taustalle piirretään väriliuku.

4.9. Piirtämisen optimointi kaksoispuskuroinnilla

Räätälöidyn kontrollin piirtoa voidaan optimoida tekniikalla, jonka nimi on kaksoispuskurointi (double buffering). Kaksoispuskuroinnin ajatuksena on vähentää askelten määrää, joilla piirtäminen tapahtuu. Esimerkiksi piirtokerran aikana voi tapahtua kaksi piirtoaskelta, joissa ensimmäisessä piirretään neliö ja toisessa ympyrä. Tällöin käyttäjä voi ehtiä havaitsemaan, että .NET-käyttöliittymä piirsi ensin neliön ja vasta sitten ympyrän. Piirtoaskelten välinen lyhyt viive näkyy käyttäjälle sitä selvemmin, mitä enemmän piirtoaskelia piirtokerrassa on.

Kaksoispuskurointi korvaa piirtokerran kaikki piirtoaskelet yhdellä piirtoaskelella. Kaksoispuskurointi toimii siten, että piirtokerta suorittaa piirtoaskelet samoin kuin ennenkin, mutta ne piirtävätkin näytön piirtopinnan sijaan muistin piirtopinnalle eli muistissa olevaan bittikarttaan. Vasta, kun piirtokerran kaikki piirtoaskelet on suoritettu, bittikartta piirretään näytön piirtopinnalle yhdellä piirtoaskelella. Tällöin käyttäjä näkee suoraan lopputuloksen eikä vaiheita, joiden kautta lopputulos saatiin aikaan.

Kaksoispuskurointia käytettäessä piirtämiseen käytetty aika säilyy samana. Kaksoispuskurointi siis ainoastaan viivästää seuraavaksi näkyville tulevan sisällön näyttämisen aina hetkeen, kun sisältö on kokonaan valmis. Tämän takia piirtokoodi kannattaa optimoida myös kaksoispuskurointia käytettäessä.

Windows Forms tarjoaa valmiin kaksoispuskuroinnin, joka on kontrollikohtaista. Koska kontrollit eivät jaa samaa valmiiseen kaksoispuskurointiin käytettävää bittikarttaa, käyttäjä voi valmiista kaksoispuskuroinnista huolimatta nähdä, miten kontrollit piirtyvät vuorotellen. Tällöin jokainen kontrolli, johon valmis kaksoispuskurointi on otettu käyttöön, piirtää ainoastaan itsensä yhdellä piirtoaskelella.

Valmis kaksoispuskurointi poistaa piirron välkkymisen siis kontrollikohtaisesti, muttei kontrollien väliltä. Tämä ilmiö näkyy käyttäjälle siten, että käyttöliittymäikkuna piirtää itsensä yhdellä piirtoaskelella, sen sisällä oleva jokin kontrolli piirtää itsensä yhdellä piirtoaskelella, jokin seuraava kontrolli piirtää itsensä yhdellä piirtoaskelella ja niin edelleen. Jokainen kaksoispuskuroitu kontrolli viivästää seuraavaksi näkyville tulevan sisällön näyttämistä, ja viivästetty aika on sama kuin aika, jonka käyttäjä ehtii huomata kontrollien piirtymisen välissä. Vuorotellen piirtyviä kontrolleja voidaan välttää optimoimalla niiden piirtokoodia.

Räätälöity kontrolli ottaa valmiin kaksoispuskuroinnin käyttöön eri tavoin riippuen metodeista, jotka vastaavat piirtokertaa. Piirtokerta tapahtuu joko metodeissa *OnPaintBackground* ja *OnPaint* tai metodissa *OnPaint*. Molemmissa tapauksissa val-

miin kaksoispuskuroinnin käyttöön ottaminen tapahtuu tavallisesti räätälöidyn kontrollin rakentajassa.

Jos piirtokerta tapahtuu molemmissa metodeissa, räätälöity kontrolli ottaa valmiin kaksoispuskuroinnin käyttöön kutsumalla metodiaan *SetStyle* parametrilla, jossa on ainoastaan arvo *ControlStyles.OptimizedDoubleBuffer*. Räätälöidyllä kontrollilla on olemassa myös ominaisuus *DoubleBuffered*. Tässä tilanteessa kyseistä ominaisuutta ei pidä asettaa arvoon tosi, koska muutoin, lähteen [3, s. 248] mukaan, räätälöity kontrolli ei välttämättä pyyhi taustaansa esiin oikein, kun käyttäjä vaihtaa ohjelmasta toiseen näppäinten Alt ja Sarkain avulla.

Jos piirtokerta tapahtuu taas ainoastaan metodissa *OnPaint*, räätälöity kontrolli ottaa valmiin kaksoispuskuroinnin käyttöön asettamalla ominaisuutensa *DoubleBuffered* arvoon tosi. Koska tällöin räätälöity kontrolli ei pyyhi taustaansa esiin, metodin *OnPaint* on piirrettävä räätälöidyn kontrollin koko piirtoalueelle.

Kun valmis kaksoispuskurointi on otettu käyttöön räätälöidylle kontrollille, valmis kaksoispuskurointi ei näy ohjelmoijalle mitenkään eli räätälöity kontrolli piirtää itsensä samoin kuin ennenkin. Räätälöity kontrolli saa siis yhä käyttöönsä tyyppiä *Graphics* olevan olion. Tällöin olio kapseloikin näytön piirtopinnan sijaan muistin piirtopinnan eli muistissa olevan bittikartan. Windows Forms huolehtii siitä, piirtokerran piirtoaskelten päätyttyä kyseinen bittikartta piirretään näytön piirtopinnalle yhdellä piirtoaskelella.

Valmista kaksoispuskurointia ei kannata käyttää, jos suurin osa räätälöidyn kontrollin piirtoalueesta säilyy piirtokertojen välillä muuttumattomana. Valmiin kaksoispuskuroinnin käyttäminen piirtoalueen pientä aluetta varten aiheuttaa nimittäin muistille ylikuormaa, koska valmiin kaksoispuskuroinnin bittikartta kattaa koko piirtoalueen. Tällöin muistin kannalta on parempi ratkaisu kaksoispuskuroida pieni alue käsin. Tämä tapahtuu siten, että piirtokerta piirtää pienen alueen usealla piirtoaskelella muuttujana olevaan bittikarttaan, jonka piirtokerta piirtää lopuksi näytön piirtopinnalle yhdellä piirtoaskelella.

.NET-rajapinnan luokka *System.Drawing.Bitmap* vastaa bittikarttaa, johon voidaan piirtää. Piirtämistä varten sille on luotava piirtopinta, mikä tapahtuu staattisella metodilla *Graphics.FromImage*.

4.10. Piirtämisen optimointi välimuistilla

Räätälöidyn kontrollin piirtoa optimoidaan välimuistilla siten, että räätälöity kontrolli tallettaa piirtämänsä sisällön kuvina välimuistiin myöhempiä piirtokertoja varten. Näin räätälöity kontrolli voi välttää aikaa vievää piirtokoodia seuraavilla piirtokerroilla. Välimuistin kuvat voivat esimerkiksi olla räätälöidyn kontrollin eri tiloista ja räätälöidyn kontrollin koko piirtoalueen kokoisia. Välimuistin käytöstä on käytännössä hyötyä, jos piirtokoodi vie aikaa ja hitaus on havaittavaa.

Välimuistin käyttö vähentää piirtoon kuluvaa aikaa muistin kustannuksella. Jos välimuistia käyttävästä räätälöidystä kontrollista tehdään tyypillisesti monta oliota, kannattaa harkita olioiden yhteisen välimuistin toteuttamista, jotta muistia säästyy.

Välimuistin käyttö voi olla ratkaisu kohdassa 4.9 kuvattuun valmiin kaksoispuskuroinnin ongelmaan, jossa kontrollit piirtyvät näkyvästi vuorotellen. Kyseisessä kohdassa todettiin, että ratkaisu on piirtokoodin optimointi, ja välimuistin käytöllä saatetaan optimoida piirtokoodia riittävästi.

5. KONTROLLIHIERARKIAN TOIMINTA

Tässä luvussa perehdytään lisää kontrollihierarkiaan ja sen toimintaan keskittyen lähinnä rajoitteisiin ja kontrollien ulkoasuun. Kohdassa 5.1 esitellään kontrollihierarkian rajoitteet sen sisältöön ja rakenteeseen liittyen ([3, s. 10]). Kohdassa 5.2 otetaan esille taas kontrollien määrään liittyvät rajoitteet ([8; 9; 10]). Kohdassa 5.3 tarkastellaan, miten muodostettu kontrollihierarkia saatetaan näkyville .NET-käyttöliittymään.

Kohdassa 5.4 tutustutaan kontrollin ketjutettuihin ominaisuuksiin *Visible* ja *Enabled* sekä kohdassa 5.5 taas kontrollin ympäröiviin ominaisuuksiin ([3, s. 54, s. 339, s. 429]), joita räätälöity kontrolli voi käyttää itsensä piirtämisessä. Sekä ketjutetut että ympäröivät ominaisuudet ovat kontrollihierarkiasta riippuvaisia.

Kohdassa 5.6 esitellään kontrollihierarkian kontrollien piirtojärjestys ([3, s. 50–51]). Kontrollin piirtoalueeseen ja kontrollin piirtokerran piirtoalueeseen keskitytään taas kohdassa 5.7. Seuraavissa kohdissa 5.8 ja 5.9 tarkastellaan sitä, miten kontrolli voi menettää piirtoaluettaan eri tavoin (kohdassa 5.9 [3, s. 75, s. 819–822]).

Kohdassa 5.10 perehdytään käyttöliittymäikkunan sovittamiseen taustakuvaansa aiemmin esiteltyjen toiminnallisuuksien avulla ([3, s. 815–822]). Lopuksi kohdissa 5.11 ja 5.12 käsitellään eri tavoin toteutettuja läpinäkyvyyksiä kontrollihierarkian kontrollien välillä (kohdassa 5.11 [3, s. 54–55, s. 838–839]).

5.1. Rajoitteet sisällössä ja rakenteessa

Kontrollihierarkian voi muodostaa valmiita kontrolleja ja räätälöityjä kontrolleja vapaasti yhdistellen, koska minkä tahansa kontrollin voi laittaa toisen sisälle kolmea poikkeustilannetta lukuun ottamatta. Ensimmäisessä poikkeustilanteessa sisälle laitettava kontrolli on jo entuudestaan jonkun toisen kontrollin sisällä, jolloin vanha sisältyvyys poistetaan automaattisesti ennen uuden muodostamista. Näin varmistetaan, että kukin kontrolli on kerrallaan enintään yhden toisen kontrollin sisällä, mistä mainittiin aiemmin kohdassa 2.2.

Toisessa poikkeustilanteessa muutos muodostaisi kehärakenteen, josta mainittiin aiemmin kohdassa 2.2. Tällöin muutos estyy, ja muutosta varten käytetty ominaisuus tai metodi heittää poikkeuksen. Kolmannessa poikkeustilanteessa muutoksen seurauksena käyttöliittymäikkuna päätyisi kontrollin sisälle. Tällöinkin muutos estyy, ja heitetään poikkeus.

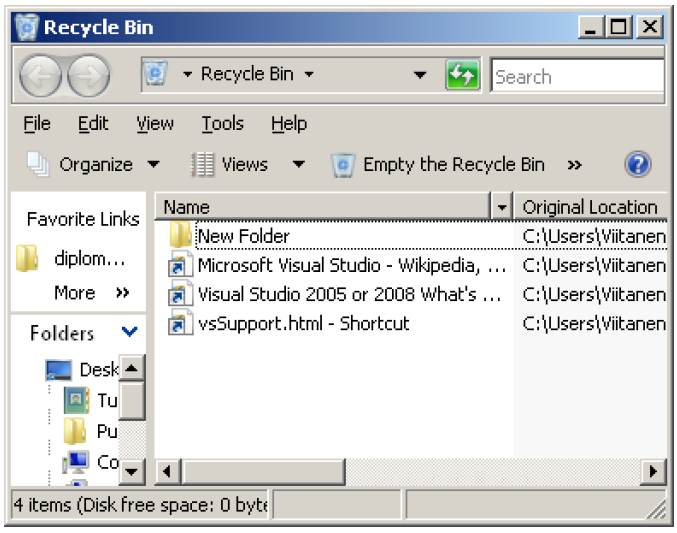
5.2. Rajoitteet määrässä

Kontrollihierarkia ei itsessään rajoita siihen laitettavien kontrollien määrää. Sen sijaan kontrollien kokonaismäärää rajoitetaan Windowsin istuntoa kohti, minkä seurauksena kontrollien kokonaismäärää rajoitetaan myös prosessia kohti, jottei yksi prosessi voi saada käyttöönsä kaikkia resursseja.

Kontrolleja voi olla Windowsin versioissa XP ja Vista istuntoa kohti enimmillään noin 32 700. Molempien versioiden prosessikohtainen raja on oletuksena 10 000 kontrollia [11], mutta raja voidaan muuttaa Windowsin rekisteristä välille 200 – 18 000. Jos prosessi yrittää tehdä kontrolleja enemmän kuin on mahdollista, Windows tulkitsee prosessin käyttäytyvän epäsopivasti ja sulkee sen. Juuri tämän takia esimerkiksi luettelonäkymää kuvaavan valmiin kontrollin *ListView* alkiot eivät ole kontrolleja: 100 alkiota leveä ja 200 alkiota korkea taulukko ei olisi mahdollinen prosessikohtaisen rajan muuttamisesta huolimatta (kuva 4.1 sivulla 14).

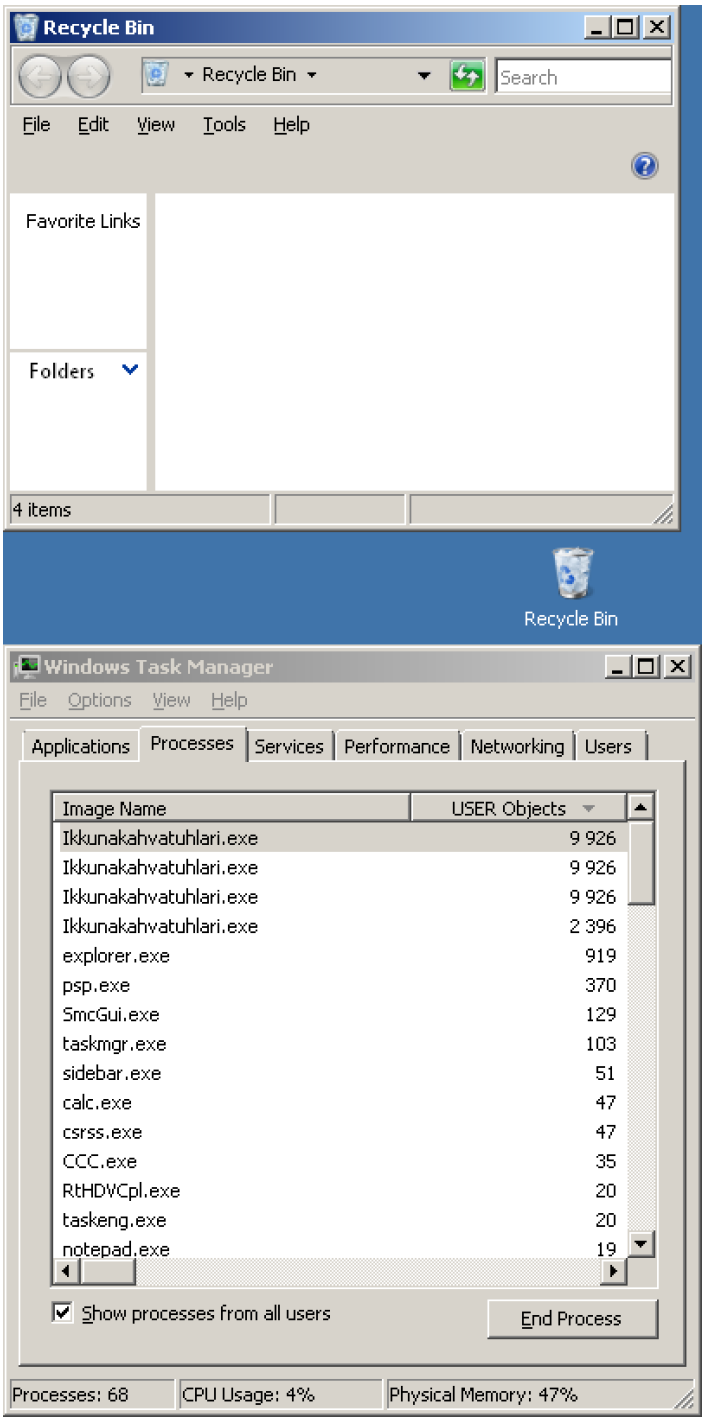
Windows Forms rajoittaa kontrollien kokonaismäärää, koska Windows API -rajapinta rajoittaa ymmärtämiensä kontrollien kokonaismäärää ja Windows Forms ei onnistu piilottamaan tätä rajoitetta. Windows API -rajapintaa ei ole aikoinaan suunniteltu sellaista kokonaismäärää varten, johon vaikuttaa ainoastaan saatavilla olevan muistin määrä.

Windows API -rajapinta tunnistaa kunkin kontrollin ymmärtämässään muodossa ikkunakahvojen (windows handle) perusteella, jotka ovat kokonaislukuja. Jokaiseen kontrolliin liittyy yksi ikkunakahva kontrollin alustamisen jälkeen. Ikkunakahvat ovat alityyppi sellaisille kahvoille, joiden kokonaismäärä on rajoitettu Windowsin istuntoa kohti teoreettisesti 65 536:een. Enimmäismäärä on käytännössä kuitenkin vain puolet teoreettisesta arvosta, koska ikkunakahvat ovat parillisia lukuja, mihin monet ohjelmat luottavat [12].



Kuva 5.1. *Roskakori Windows Vistassa, josta visuaaliset tyylit on otettu pois käytöstä*

Windowsin istuntoa kohti rajoitetun kontrollien kokonaismäärän havainnollistamiseksi tehtiin kokeiluohjelma, joka varaa kontrolleja prosessikohtaista rajaa varoen. Ennen kokeiluohjelman käynnistämistä Windowsin roskakori näytti kuvan 5.1 mukaiselta. Roskakorissa oli neljä kohdetta, ja roskakorin käyttämät kontrollit näkyivät oikein.



Kuva 5.2. *Roskakorille ei riitä ikkunakahvoja*

Seuraavaksi kontrolleja varaava kokeiluohjelma käynnistettiin neljään kertaan sopivilla kontrollien määrillä. Näin toimimalla päästiin lähelle suurinta mahdollista kontrollien kokonaismäärää kyseisessä Windowsin istunnossa. Tämän jälkeen roskakori avat-

tiin, mutta se ei onnistunutkaan luomaan kaikkia käyttämiään kontrolleja, mikä näkyy kuvassa 5.2. Roskakorissa oli edelleen neljä kohdetta, vaikkeivät ne virheellisesti olleetkaan enää näkyvillä.

Kuvassa näkyy myös Tehtävienhallinta, jonka sarakkeessa USER Objects näkyy prosessien varaamien ikkunakahvojen määrät. Sarake kuvaa kohteita sellaisille kahvoille, joiden alityyppinä on ikkunakahvat. Kokeilun aikana havaittiin myös, että Windows sulki kokeiluohjelman, jos se yritti luoda kontrolleja liikaa.

5.3. Näyttäminen ja muutokset

Kontrollihierarkia, jonka juurena on käyttöliittymäikkuna, saatetaan näkyville .NET-käyttöliittymään kutsumalla käyttöliittymäikkunan metodia *Show* tai *ShowDialog* tai antamalla käyttöliittymäikkuna parametrina metodille *System.Windows.Forms.Application.Run*. Se säilyttää ohjelman käynnissä ilman eri toimenpiteitä, koska suoritus palaa kyseiseltä metodilta vasta, kun käyttöliittymäikkuna suljetaan. [3, s. 31–33.] Näkyville ei voida saattaa yksittäisiä kontrolleja eikä kontrollihierarkiaa, jonka juurena ei ole käyttöliittymäikkuna.

Kun kontrollihierarkia on saatettu näkyville, sen rakennetta voidaan edelleen muokata. Jos kontrollihierarkiaan lisätään uusi kontrolli, se ja kaikki sen sisällä olevat kontrollit ilmestyvät näkyville käyttöliittymäikkunaan. Jos kontrollihierarkiasta taas poistetaan kontrolli, vastaavasti se ja kaikki sen sisällä olevat kontrollit poistuvat näkyvistä.

5.4. Ketjutetut ominaisuudet

Ketjutetut ominaisuudet ovat kontrollin ominaisuuksia, jotka riippuvat kontrollihierarkian kautta muiden kontrollien vastaavan ominaisuuden arvosta. Ketjutettu ominaisuus on tyypiltään aina totuusarvo, ja ketjutetun ominaisuuden yhteydessä on luontevaa ajatella rekursiivisesti siten, että, jos arvo on epätosi, myös jokaisen sisällä olevan kontrollin vastaava arvo näkyy ulospäin epätotena.

Kontrollin ominaisuuksista *Visible* ja *Enabled* ovat ketjutettuja. Ominaisuus *Visible* määrittää, onko kontrolli näkyvillä. Ominaisuus *Enabled* puolestaan määrittää, onko kontrolli käytettävissä. Jos kontrolli ei ole käytettävissä, sen käyttö näppäimistöllä ja hiirellä on estetty, mistä kontrolli tavallisesti ilmoittaa ulkoasullaan (kuva 6.1 sivulla 40).

Ominaisuuden *Visible* on luontevaa olla ketjutettu, koska kontrollin ollessa näkymättömissä minkään sen sisälläkään olevan kontrollin ei ole mielekästä näkyä. Ominaisuuden *Enabled* kohdalla voidaan ajatella vastaavasti.

Ketjutetun ominaisuuden asettaminen todeksi epäonnistuu, jos sisältävän kontrollin sama ominaisuus on epätosi. Tällöin ominaisuus vain säilyy muuttumattomana eikä esimerkiksi heitä poikkeusta. Jos ketjutetun ominaisuuden asettaminen todeksi onnistuu, jokaisen sisällä olevan kontrollin sama ominaisuus voi myös muuttua todeksi: ketjutettu ominaisuus muistaa arvon, joka viimeksi on yritetty asettaa ketjutettuun ominaisuuteen,

ja ketjutettu ominaisuus alkaa esittää tätä arvoa, kun sisältävän kontrollin sama ominaisuus muuttuu todeksi.

Ketjutetun ominaisuuden asettaminen epätodeksi onnistuu aina. Tällöin jokaiselle sisälle olevalle kontrollille tulee sama arvo samaan ominaisuuteen.

5.5. Ympäröivät ominaisuudet

Ympäröivät ominaisuudet (ambient properties) ovat kontrollin ominaisuuksia, jotka riippuvat kontrollihierarkian kautta muiden kontrollien vastaavan ominaisuuden arvosta. Ympäröivän ominaisuuden yhteydessä on luontevaa ajatella rekursiivisesti siten, että, jos arvoa ei ole asetettu, arvoksi tulkitaan sisältävän kontrollin vastaava arvo ja se näkyy kyseisen ominaisuuden arvona. Ympäröivien ominaisuuksien tarkoitus on mahdollistaa sisällä olevien kontrollien ulkoasun sulautuminen ympäristöönsä vaivattomasti. Ilman ympäröiviä ominaisuuksia samat arvot olisi asetettava jokaiseen kontrolliin erikseen, jotta käyttöliittymäikkuna näyttää yhtenäiseltä.

Kontrollin ominaisuuksista *BackColor*, *ForeColor*, *Font*, *Cursor* ja *RightToLeft* ovat ympäröiviä. Ominaisuus *BackColor* määrittää kontrollin taustalla käytetyn värin ja *ForeColor* taas etualalla käytetyn värin, *Font* määrittää kontrollin käyttämän kirjasinlajin, tekstikoon ja tekstityylin, *Cursor* määrittää kontrollin yläpuolella olevan hiiren kohdistimessa käytetyn kuvan ja *RightToLeft* määrittää, tasataanko kontrollin sisältö tukemaan lokaaleja, jotka käyttävät oikealta vasemmalle luettavia kirjasinlajeja. Räätälöity kontrolli voi käyttää näitä ominaisuuksia piirron aikana huolehtimatta siitä, onko jokainen asetettu räätälöidylle kontrollille.

Ympäröivien ominaisuuksien toteuttamisessa on käytetty apuna luokkaa *System.Windows.Forms.AmbientProperties*. Sen yhteyteen on dokumentoitu asettamattoman ympäröivän ominaisuuden arvo tilanteessa, jossa samaa ympäröivää ominaisuutta ei ole asetettu millekään sisältävälle kontrollille.

5.6. Piirtojärjestys

Kontrollien piirtojärjestys tarkoittaa sitä järjestystä, jossa kontrollihierarkian kontrollit piirretään. Piirtojärjestyksen tavoitteena on, että sisällä oleva kontrolli piirtyy aina sen sisältävän kontrollin jälkeen. Tällöin sisällä oleva kontrolli peittää sisältävää kontrollia eli tulee sen päälle. Piirtojärjestyksen tavoitteena on myös tarjota toteuttajalle mahdollisuus määrittää, missä järjestyksessä saman kontrollin sisällä olevat kontrollit piirretään. Tällä on merkitystä silloin, kun saman kontrollin sisällä olevat kontrollit ovat osin päällekkäin.

Edellä mainittujen tavoitteiden saavuttamiseksi piirtojärjestys on määritelty kontrollihierarkiaan perustuen. Piirtojärjestyksessä alimmainen on käyttöliittymäikkuna ja päällimmäisiä ovat kontrollihierarkiassa lehtinä olevat kontrollit. Jos jonkun kontrollin sisällä on useita kontrolleja, piirtojärjestys määräytyy niiden välillä siitä järjestyksestä, jossa ne ovat kyseisen kontrollin sisällä. Tästä järjestyksestä käytetään nimeä z-järjestys (z-

order). Z-indeksi (z-index) puolestaan tarkoittaa järjestysnumeroa, joka kuvaa, monentenako kontrolli on sisältävän kontrollin sisällä. Z-indeksit aloittavat jokaisen kontrollin sisällä arvosta 0, joka tarkoittaa piirtojärjestyksessä päällimmäistä kontrollia.

Kuvassa 5.3 on kontrollihierarkia, joka esitetään kuvassa puumaisena rakenteena, joka kuvaa kontrollien väliset suhteet sekä järjestyksen, jossa kontrollit ovat toistensa sisällä. Z-indeksi on 0 kontrollin sisällä ensimmäisenä olevalla kontrollilla eli kuvassa aina ylimpänä olevalla kontrollilla. Esimerkiksi valmiiden kontrollien *laatikko1* ja *vaihtoehto1* z-indeksi on 0. Kuvan kontrollihierarkiasta näkyy myös, että valmiin kontrollin *laatikko1* sisällä on kaksi valmista kontrollia. Samoin käyttöliittymäikkunan sisällä on kaksi valmista kontrollia, joista valmiin kontrollin *painike1* z-indeksi on 1.

Kuva 5.3. *Kontrollihierarkia puumaisena rakenteena esitettynä*

Jos saman kontrollin sisällä olevat kontrollit ovat osin päällekkäin, niiden keskinäinen piirtojärjestys määräytyy z-järjestyksestä. Esimerkiksi kuvan valmis kontrolli *laatikko1* piirtyy osin valmiin kontrollin *painike1* päälle, jos ne ovat osin päällekkäin.

Windows Forms piirtää kontrollihierarkian jokaisen kontrollin noudattaen piirtojärjestystä. Windows Forms siis kutsuu näkyvillä olevien kontrollien metodeja *OnPaintBackground* ja *OnPaint* siten, että lopputuloksena kontrollit näyttävät olevan sisäkkäin tarkoituksenmukaisessa järjestyksessä. Kuvassa 5.4 näkyy kuvan 5.3 kontrollihierarkia, jonka Windows Forms piirsi piirtojärjestyksen mukaisesti. Kuvassa 5.4 jotkin kontrollit ovat osin päällekkäin, mikä paljastaa piirtojärjestyksen saman kontrollin sisällä olevien kontrollien välillä.

Kuva 5.4. *Kuvan 5.3 kontrollihierarkia Windows Formsin piirtämänä*

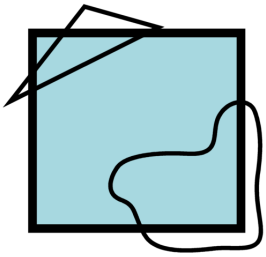
Piirtojärjestykseen voidaan vaikuttaa kontrollihierarkiaa muuttamalla. Z-järjestys saadaan erilaiseksi vaihtamalla järjestystä, jossa kontrollit lisätään. Sen jälkeen, kun kontrollit on jo lisätty, kontrollin z-indeksiä voidaan muuttaa esimerkiksi kontrollin metodeilla *BringToFront* ja *SendToBack* sekä sisältävän kontrollin ominaisuuden *Controls* tarjoaman kokoelman metodilla *SetChildIndex*. Se asettaa z-indeksin parametrinaan saamalleen kontrollille. Z-indeksin arvon voi hakea saman kokoelman metodilla *GetChildIndex*.

Kontrollin lisääminen toisen kontrollin sisälle toimii siten, että lisätyn kontrollin z-indeksiksi tulee suurin arvo. Tästä seuraa, että viimeisenä samaan kontrolliin lisätty kontrolli on piirtojärjestyksessä alimmainen. Tämä ei ole välttämättä toteuttajalle intuitiivista. Ratkaisun syynä saattaa olla, että z-järjestyksen ylläpitäminen on näin tehokkaampaa: uuden kontrollin lisäämisen yhteydessä ei ole muutettava jokaisen vanhan kontrollin z-indeksiä, jotta lisätyn kontrollin z-indeksiksi saadaan 0. Esimerkiksi lähteessä [3, s. 864] on päädytty toteuttamaan räätälöidyn kontrollin sisälle oma piirtojärjestys, jossa on valittu intuitiivisuus tehokkuuden kustannuksella.

5.7. Piirtoalue

Kuten aiemmin todettiin, kontrollin piirtoalue on se alue, jolle kontrolli voi piirtää. Kontrolli on siis läpinäkyvä piirtoalueensa ulkopuolisilta osilta. Koska kontrollit piirretään piirtojärjestyksen mukaisesti, kukin kontrolli on läpinäkyvä piirtoalueensa ulkopuolelta siihen kontrolliin, joka on piirtojärjestyksen mukaan seuraavana alapuolella. Piirtoalue määräytyy kontrollin ominaisuuksista *Bounds* ja *Region*.

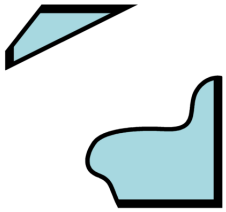
Ominaisuus *Bounds* määrittää kontrollin sijainnin sisältävän kontrollin koordinaateissa sekä leveyden ja korkeuden eli suorakulmaiset rajat, joiden ulkopuolelle tehdyllä piirrolla ei ole merkitystä. Ominaisuus *Region* puolestaan voi pienentää tätä aluetta, muttei suurentaa. Ominaisuus *Region* on mielivaltainen kokoelma pikseleitä, jotka kuvaavat aluetta, jolla kontrollin on tarkoitus näkyä. Tästä seuraa, että piirtoalue voi olla epäyhtenäinen. Esimerkiksi kuvassa 5.5 on kontrolli, jonka ominaisuutta *Bounds* kuvaa paksumpi viiva. Kontrollin ominaisuutta *Region* kuvaa taas ohuempi viiva ulottuen osin ominaisuuden *Bounds* määrittämien rajojen ulkopuolelle.



Kuva 5.5. *Kontrollin ominaisuudet Bounds ja Region, joista jälkimmäinen on ohuemmalla viivalla ja epäyhtenäinen*

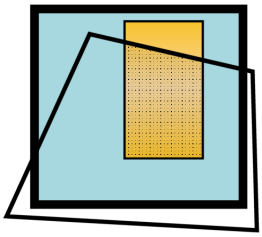
Kuvassa 5.6 näkyy kuvan 5.5 piirtoalue, joka muodostuu ominaisuuksien *Bounds* ja *Region* leikkauksesta. Esimerkkinä käytetty kontrolli on läpinäkyvä kuvan 5.6 valkoisella alueella.

Vaikka piirtoalue onkin se alue, jolle kontrolli voi piirtää, kontrolli ei voi aina piirtää koko piirtoalueelleen siten, että kaikki piirretty päätyy näkyville. Kuten kohdassa 4.5 mainittiin, kontrolli voi piirtää piirtokertansa aikana ainoastaan piirtoalueensa niille osille, jotka ovat epäkelpoja. Tästä seuraa, että kontrollin piirtokerran piirtoalue voi olla pienempi kuin kontrollin piirtoalue.



Kuva 5.6. *Kuvan 5.5 piirtoalue*

Kuten kohdassa 4.8 todettiin, piirtämisen avuksi tarjottavan luokan *PaintEventArgs* ominaisuus *ClipRectangle* kattaa piirtoalueen kaikki epäkelvot osat. Toisin sanoen ominaisuus *ClipRectangle* määrittää piirtoalueesta suorakulmaisen osan, jonka ulkopuolelle tehdyllä piirrolla ei ole merkitystä kyseisellä piirtokerralla. Piirtokerran piirtoalue on siis kahden suorakulmion sekä yhden pikselikokoelman leikkaus, mikä näkyy kuvassa 5.7. Mikään kontrollin tämän leikkauksen ulkopuolelle piirtämä sisältö ei voi päätyä näkyville kyseisen piirtokerran aikana.



Kuva 5.7. *Kontrollin ominaisuudet Bounds ja Region sekä ohuimmalla viivalla esitetty, luokan PaintEventArgs ominaisuus ClipRectangle muodostavat piirtokerran piirtoalueen, joka on täytetty pisteillä*

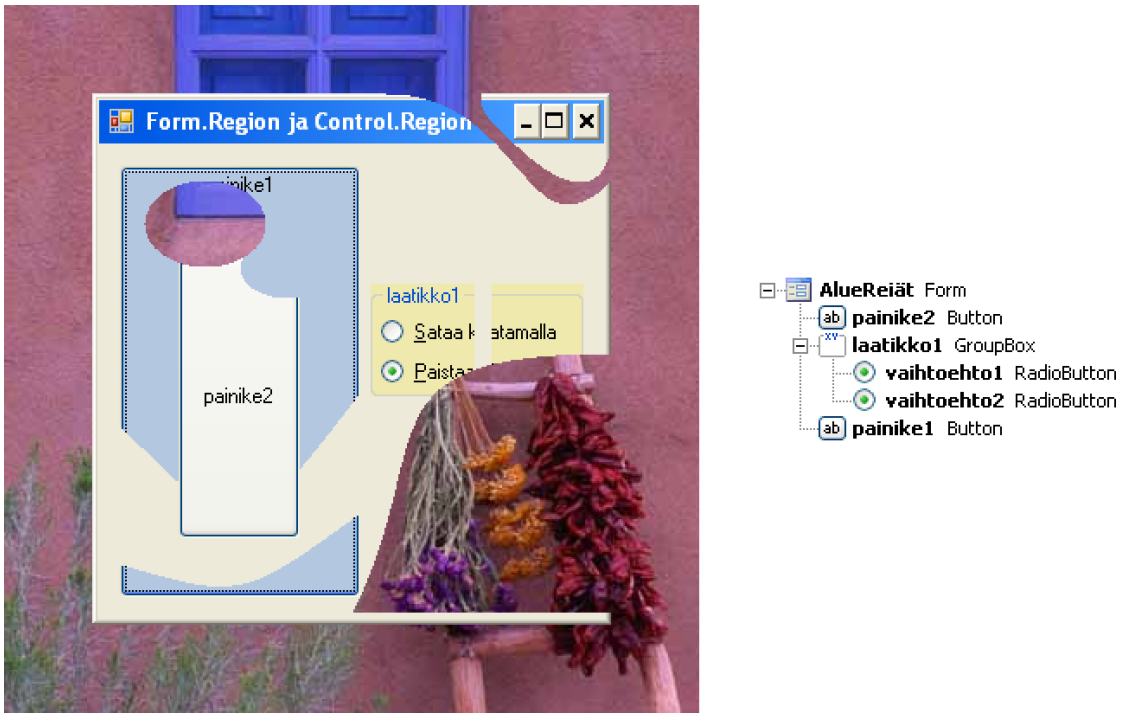
5.8. Piirtoalueen menettäminen kontrollihierarkialle

Piirtojärjestyksessä päällimmäiset kontrollit voivat peittää alimmaisia kontrolleja, jolloin ne menettävät piirtoaluettaan kontrollirakenteen vuoksi. Menetetty piirtoalue ei tällöin pääse näkyviin. Myös piirtojärjestyksessä alimmat kontrollit voivat aiheuttaa tilanteen, jossa päällimmäiset kontrollit menettävät piirtoaluettaan kontrollirakenteen välityksellä, mitä esitellään seuraavaksi tarkemmin. Tämän kohdan lopussa käsitellään sitä, miten piirtoalueen menettäminen vaikuttaa hiiren käyttöön.

Koska kontrolli ei voi piirtää piirtoalueensa ulkopuolelle, mikään sen sisällä oleva kontrollikaan ei voi piirtää saman alueen ulkopuolelle, vaikka sisällä olevan kontrollin piirtoalue ulottuisikin niin laajalle. Kontrolli voi siis näkyä vain sisältävän kontrollin piirtoalueen paljastamilta osin. Tästä seuraa, että piirtojärjestyksessä alimpien kontrollien sisällä olevat kontrollit menettävät piirtoaluettaan, jos niiden piirtoalue ulottuu sisältävän kontrollin piirtoalueen ulkopuolelle.

Kuvassa 5.8 näkyy valmis kontrolli *laatikko1*, jonka piirtoalueesta on tehty epäyhtenäinen rajaamalla keskeltä pois kapea suorakulmion muotoinen osa ominaisuuden

Region avulla. Tällöin sisällä olevat kaksi valmista kontrollia menettävät piirtoalueestaan kapean osan. Valmiin kontrollin *laatikko1* piirtoalueesta rajatun osan kohdalla näkyy nyt valmiin kontrollin *laatikko1* ja sen sisällä olevien kontrollien sijaan käyttöliittymäikkunaa johtuen siitä, että kontrollit ovat läpinäkyviä piirtoalueensa ulkopuolelta piirtojärjestyksessä seuraavana alapuolella olevaan kontrolliin, mistä mainittiin kohdassa 5.7.



Kuva 5.8. *Käyttöliittymäikkunan ja sen kontrollien ominaisuutta Region on muutettu, jolloin piirtoalueista rajautuu osia pois*

Jos kuvan valmis kontrolli *vaihtoehto1* olisi edelleen valmiin kontrollin *laatikko1* sisällä, mutta sijaintia olisi muutettu niin, ettei valmiin kontrollin *vaihtoehto1* piirtoalue olisi miltään osin sisältävän kontrollin piirtoalueen sisällä, valmis kontrolli *vaihtoehto1* ei näkyisi lainkaan. Kontrollit voivat näkyä vain sisältävän kontrollin piirtoalueen paljastamilta osin, kuten aiemmin todettiin.

Kuvassa näkyy myös käyttöliittymäikkuna, jonka piirtoalueesta on rajattu pois soikio vasemmalta ylhäältä ja kulmia oikealta ominaisuuden *Region* avulla. Käyttöliittymäikkunan piirtoalue on epäyhtenäinen, koska oikealla ylhäällä olevat painikkeet ovat erillään. Piirtoalueen rajaamisen seurauksena käyttöliittymäikkunan sisällä olevat valmiit kontrollit *painike1*, *painike2*, *laatikko1* ja *vaihtoehto2* näkyvät vain käyttöliittymäikkunan piirtoalueen paljastamilta osin. Kyseisten sisällä olevien valmiiden kontrollien menettämällä piirtoalueella näkyy nyt Windowsin työpöytää. Taustalla voisi näkyä myös jotain toista käyttöliittymäikkunaa.

Kun kontrolli menettää piirtoaluettaan kontrollihierarkialle, jäljelle jäänyt piirtoalue toimii samoin kuin ennenkin. Tämä koskee myös hiiren käyttöä. Sen sijaan menetetty piirtoalue on sellaista, jossa hiirtä ei voida käyttää kyseistä kontrollia varten. Tämä joh-

tuu siitä, että kontrolli voi käsitellä hiiritapahtumia ainoastaan sellaisessa piirtoalueessaan, jota kontrolli ei ole menettänyt.

Hiiritapahtumien toiminnasta johtuen kontrollin rei'istä voi siis painaa hiirellä niistä paljastuvaa kontrollia. Samoin käyttöliittymäikkunan reikien läpi voi painaa hiirellä Windowsin työpöytää tai takana olevaa toista käyttöliittymäikkunaa. Hiirellä painaminen kohdistuu juuri siihen kontrolliin, joka hiiren alla kulloinkin näkyy.

Kuvassa on valmiit kontrollit *painike1* ja *vaihtoehto1*, jotka näyttävät epäyhtenäisiltä. Näistä ensimmäisen piirtoalue on rajattu epäyhtenäiseksi ja jälkimmäinen on taas menettänyt piirtoaluettaan kontrollihierarkialle. Niiden molempien puolien painaminen aiheuttaa tavallisen toiminnallisuuden, ja epäjatkuvuuskohtien painaminen kohdistuu käyttöliittymäikkunaan. Kuvan valmiin kontrollin *painike2* piirtoalueesta on sen sijaan rajattu pois oikea yläkulma, jonka painaminen kohdistuu nyt valmiiseen kontrolliin *painike1*. Valmis kontrolli *painike2* ei menetä piirtoaluettaan valmiin kontrollin *painike1* epäjatkuvuuskohdan takia, koska painikkeet ovat päällekkäin, eivät sisäkkäin.

Sivuhuomiona voidaan todeta, että käyttöliittymäikkunan piirtoalueen pienentäminen ominaisuudella *Region* vaikuttaa toimivan siten, että samalla käyttöliittymäikkunan otsikkopalkin (title bar) visuaaliset tyylit poistuvat käytöstä, mikä näkyy kuvassa. Tämä saattaa liittyä siihen, että Windows XP rajaa oletuksena käyttöliittymäikkunoiden yläkulmat pyöreiksi (vrt. kuva 5.9).

5.9. Käyttöliittymäikkunan piirtoalueen menettäminen värille

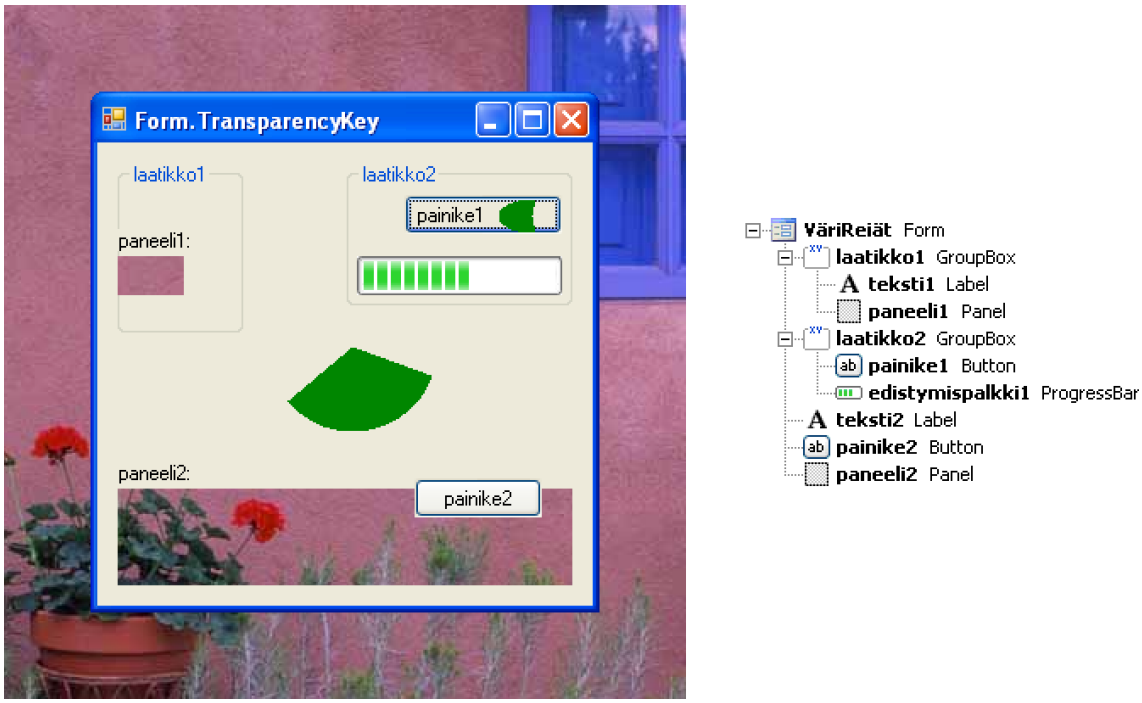
Käyttöliittymäikkuna voi menettää piirtoaluettaan kontrollihierarkialle peittyessään osin muiden kontrollien alle. Koska käyttöliittymäikkunalla ei ole koskaan sisältävää kontrollia, käyttöliittymäikkuna ei voi menettää piirtoaluettaan kontrollihierarkialle sisältävän kontrollin pienemmän piirtoalueen takia. Sen sijaan käyttöliittymäikkuna voi menettää piirtoaluettaan läpinäkyväksi asetetulle värille, mitä käsitellään seuraavaksi tarkemmin.

Käyttöliittymäikkuna menettää piirtoaluettaan värille, jos käyttöliittymäikkunan ominaisuus *TransparencyKey* on asetettu arvosta *System.Drawing.Color.Empty* poikkeavaan arvoon. Tällöin kaikki käyttöliittymäikkunan ja jokaisen sen kontrollihierarkian kontrollin kyseisellä värillä piirtämä sisältö määrittää ne pikselit, jotka käyttöliittymäikkuna menettää piirtoalueestaan. Läpinäkyväksi asetettu väri on siis siitä erikoinen, että se vaikuttaa käyttöliittymäikkunan piirtoalueeseen myös käyttöliittymäikkunan sisällä olevien kontrollien piirtämän sisällön kautta.

Seurauksena sille, että käyttöliittymäikkuna menettää piirtoaluettaan värille, käyttöliittymäikkunan kontrollihierarkian kontrollit voivat edelleen menettää omaa piirtoaluettaan kontrollihierarkialle. Lopputuloksena käyttöliittymäikkuna ja sen kontrollihierarkian kontrollit eivät näy menettämältään piirtoalueeltaan, jolloin niillä kohdin näkyy Windowsin työpöytää tai takana olevaa toista käyttöliittymäikkunaa.

Kuvassa 5.9 on käyttöliittymäikkuna, jonka ominaisuus *TransparencyKey* on asetettu vihreäksi. Käyttöliittymäikkuna piirtää keskelle vihreän viuhkan ja valmiin kontrollin

painike1 tapahtumaa *Paint* seurataan siten, että sen yhteydessä piirretään vihreä kuvio oikeaan laitaan. Vaihtoehtoisesti kyseiselle painikkeelle olisi voitu tehdä räätälöity kontrolli, mutta valmis kontrolli riittää tämän esimerkin tarkoitukseen. Valmiille kontrolleille *paneeli1*, *paneeli2* ja *edistymispalkki1* on asetettu vihreä taustaväri. Kuvasta näkyy, miten käyttöliittymäikkunan värille menettämästään piirtoalueesta näkyy läpi Windowsin työpöydälle.



Kuva 5.9. *Käyttöliittymäikkunan ominaisuus TransparencyKey on asetettu vihreäksi, jolloin käyttöliittymäikkuna menettää piirtoalueettaan vihreällä piirretyn sisällön perusteella*

Vaikka kuvan valmis kontrolli *paneeli1* on valmiin kontrollin *laatikko1* sisällä, käyttöliittymäikkuna menettää silti piirtoaluettaan värille valmiin kontrollin *paneeli1* vihreän taustavärin vuoksi. Ominaisuus *TransparencyKey* siis todella toimii käyttöliittymäikkunan koko kontrollihierarkian piirtämän sisällön kanssa.

Kuvassa näkyy, miten käyttöliittymäikkunan keskellä oleva vihreä viuhka ja valmiin kontrollin *painike1* vihreä kuvio aiheuttavat sellaisen virheen, ettei käyttöliittymäikkuna menetäkään piirtoaluettaan värille näillä kohdin. Kuva on otettu Windows XP:ssä, mutta sama kokeiluohjelma toimii toisin Windows Vistassa: tällöin virhe ei enää esiinny, ja vihreän viuhkan sekä kuvion tilalla näkyy Windowsin työpöytää.

Kuvan valmiin kontrollin *edistymispalkki1* vihreä taustaväri ei vaikuta käyttöliittymäikkunan piirtoalueen värille menettämiseen, koska silloin, kun visuaaliset tyylit ovat käytössä, kyseinen valmis kontrolli käyttää Windows XP:ssä taustansa piirtämiseen valkoista taustavärinsä sijaan. Jos Windows Vistasta poistetaan visuaaliset tyylit käytöstä, valmiin kontrollin *edistymispalkki1* taustan ja reunojen kohdalla näkyy Windowsin työpöytää. Ominaisuuden *TransparencyKey* kannalta on siis vain sillä merkitystä, mitä

kontrolli päätyy oikeasti piirtämään, eikä sillä, mitä kontrolli saattaisi piirtää jossain tilanteessa.

Ne pikselit, jotka piirtyisivät ominaisuuden *TransparencyKey* värisinä, mutta peittyvät jonkun kontrollin alle, eivät vaikuta käyttöliittymäikkunan piirtoalueen menettämiseen värille. Esimerkiksi kuvassa valmis kontrolli *painike2* on osin valmiin kontrollin *paneeli2* päällä, mutta päällekkäinen kohta on sellainen, jota käyttöliittymäikkuna ei menetä piirtoalueestaan värille. Kannattaa huomata, että valmis kontrolli *painike2* ei ole valmiin kontrollin *paneeli2* sisällä vaan päällä, mikä näkyy kontrollihierarkiasta. Lisäksi jos valmis kontrolli *painike2* olisi valmiin kontrollin *paneeli2* sisällä, kyseinen painike ei voisi piirtää sisältävän kontrollin piirtoalueen ulkopuolelle samoin kuin kuvassa.

5.10. Käyttöliittymäikkunan sovittaminen taustakuvaansa

Käyttöliittymäikkunalle voidaan sen ominaisuudella *BackgroundImage* asettaa taustakuva. Käyttöliittymäikkunan sovittamisella taustakuvaansa tarkoitetaan sitä, kun käyttöliittymäikkunan piirtoalue yritetään saada täsmäämään käyttöliittymäikkunan käyttämän taustakuvan muotoon. Jos taustakuva ei ole suorakulmainen, käyttöliittymäikkunan taustakuvan sovittamisessa voidaan käyttää apuna kohdissa 5.7–5.9 kuvattuja käyttöliittymäikkunan ominaisuuksia *Region* ja *TransparencyKey*. Seuraavaksi sovittamista käsitellään tarkemmin ja verrataan näitä kahta vaihtoehtoa toisiinsa.

Kun käyttöliittymäikkunaa sovitetaan taustakuvaansa, tavallisesti halutaan, ettei käyttöliittymäikkunasta jää näkyviin mitään taustakuvan reunojen ulkopuolelle. Tällöin halutaan piilottaa käyttöliittymäikkunasta kaikki ne alueet, joihin käyttöliittymäikkunan metodilla *OnPaint* ei voi piirtää. Tällaisia alueita ovat käyttöliittymäikkunan otsikkopalkki ja reunat, joiden kohdalta käyttöliittymäikkunan kokoa voidaan muuttaa hiirellä. Nämä alueet piilotetaan asettamalla käyttöliittymäikkunan ominaisuus *FormBorderStyle* arvoon *FormBorderStyle.None*.

Jos käyttöliittymäikkuna sovitetaan taustakuvansa reunoihin ominaisuuden *Region* avulla, sille on muodostettava ohjelmallisesti käyrä, joka on taustakuvan reunan muotoinen. Tämä on työläämpää verrattuna sovittamiseen ominaisuuden *TransparencyKey* avulla, koska jälkimmäisessä vaihtoehdossa taustakuvan väri aiheuttaa automaattisesti vastaavan lopputuloksen. Toisaalta tällöin käyttöliittymäikkunan kontrollihierarkian kontrollien on käytettävä läpinäkyvästä väristä poikkeavia värisävyjä saadakseen piirtämänsä sisällön näkyviin. Nopeamman vaihtoehdon haittana ovat myös puutteet Windows XP:ssä (kohta 5.9).

Ominaisuuden *TransparencyKey* käyttäminen voi olla muillakin tavoin ongelmallista. Ominaisuus *TransparencyKey* ei nimittäin toimi joillakin näytönohjaimilla, jos värisyvyys on suurempi kuin 24-bittiä. Lisäksi ongelmia voi tulla kaksoispuskuroinnin (kohta 4.9) kanssa käytettynä. Näistä ongelmista johtuen ominaisuutta *TransparencyKey* käytettäessä kannattaa aina varmuuden vuoksi käyttää myös ominaisuutta *Region* edes karkeana arviona käyttöliittymäikkunan taustakuvan muodosta, jotta ongelmat näkyvät pienempinä. [3, s. 820.]

5.11. Simuloitu läpinäkyvyys

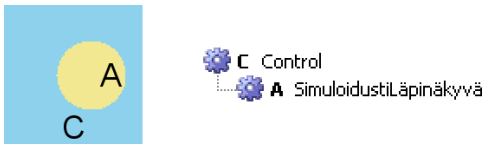
Simuloidulla läpinäkyvyydellä tarkoitetaan tekniikkaa, jolla kontrolli voi olla piirtoalueeltaan läpinäkyvä aivan kuin kontrollin piirtoaluetta olisi rajattu. Simuloitua läpinäkyvyyttä käytettäessä kontrollin piirtoaluetta ei kuitenkaan rajata. Simuloidun läpinäkyvyyden tarkoitus on helpottaa läpinäkyvyyden toteuttamista piirtoalueen rajaamiseen verrattuna. Piirtoalueen rajaaminen läpinäkyvyyttä varten on työlästä esimerkiksi tilanteessa, jossa kontrollin halutaan olevan läpinäkyvä sen piirtämien kirjainten välistä.

Simuloidusti läpinäkyvä kontrolli toimii siten, että sen kantaluokan *Control* metodi *OnPaintBackground* kopioi sisältävän kontrollin piirtämän sisällön piirtopinnalle. Kopioinnin jälkeen aliluokan metodit *OnPaintBackground* ja *OnPaint* piirtävät ne pikselit, joiden ei ole tarkoitus olla läpinäkyviä. [3, s. 55, s. 839.]

Simuloidusti läpinäkyvän räätälöidyn kontrollin metodit *OnPaintBackground* ja *OnPaint* voivat siis piirtää samoin kuin ennenkin, kunhan metodi *OnPaintBackground* kutsuu aluksi kantaluokan toteutusta. Jos räätälöity kontrolli piirtää jotain läpinäkyvällä värillä, piirtopinnan sisältö ei muutu niiltä kohdin. Jos esimerkiksi räätälöity kontrolli piirtää osin läpinäkyvän kuvan, läpinäkyvät pikselit eivät muuta piirtopintaa. Jos siis räätälöity kontrolli piirtää ensin näkyvällä värillä ja sitten samaan kohtaan läpinäkyvällä värillä, jälkimmäinen piirtäminen ei muuta piirtopintaa eikä sisältävän kontrollin piirtämän sisällön kopio paljastu enää takaisin esille.

Simuloitu läpinäkyvyys asetetaan päälle kontrolliin kutsumalla sen metodia *SetStyle* parametreilla *ControlStyles.SupportsTransparentBackColor* ja true. Tämän lisäksi kontrollin ominaisuus *BackColor* asetetaan arvoon *Color.Transparent*.

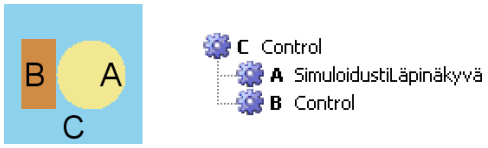
Kuvassa 5.10 on valmis kontrolli *C*, jonka sisällä on räätälöity kontrolli *A*. Se asettaa rakentajassaan läpinäkyvyyden simuloinnin päälle ja piirtää metodissaan *OnPaint* neliskulmaiselle piirtoalueelleen ympyrän jättäen kulmat piirtämättä. Tällöin kulmien kohdalla näkyy sisältävää kontrollia *C* aivan kuin kontrolli *A* piirtäisi suoraan sisältävän kontrollin *C* piirtopinnalle.



Kuva 5.10. *Simuloitu läpinäkyvyys toimii oikein ympyrän kanssa*

Simuloidun läpinäkyvyyden nimessä on sana simuloitu, koska läpinäkyvyys ei ole aitoa. Näin on, koska simuloidussa läpinäkyvyydessä piirtämättä jätetyistä piirtoalueen osista näkyy sisältävää kontrollia sen sijaan, että näkyisi piirtojärjestyksessä alapuolella olevaa kontrollia. Aito läpinäkyvyys saavutetaan esimerkiksi rajaamalla piirtoaluetta, mikä toisaalta voi olla työläämpää. Simuloitu läpinäkyvyys riittää kuitenkin moniin tilanteisiin ja on varsin käyttökelpoinen ratkaisu, kunhan tunnistaa sen rajoitteet etukäteen.

Simuloidun läpinäkyvyyden rajoitteet käyvät ilmi kuvasta 5.11. Siinä räätälöidyllä kontrollilla *A* ja valmiilla kontrollilla *B* on neliön muotoinen piirtoalue, ja *A* piirtää ympyrän ja *B* neliön. Räätälöity kontrolli *A* on simuloidusti läpinäkyvä ja puoliksi valmiin kontrollin *B* päällä.



Kuva 5.11. *Simuloitu läpinäkyvyys toimii väärin neliön ja ympyrän kanssa*

Kuvan molemmat kontrollit *A* ja *B* ovat saman kontrollin *C* sisällä, joten räätälöidyn kontrollin *A* vasemmasta laidasta näkyy virheellisesti sisältävää kontrollia *C* piirtojärjestyksessä alapuolella olevan valmiin kontrollin *B* sijaan. Se näyttää nyt virheellisesti suorakulmiolta neliön sijaan. Läpinäkyvyyden simulointi siis toimii, kunhan simuloidusti läpinäkyvä kontrolli ei ole saman kontrollin sisällä toisen kontrollin kanssa ja sen päällä.

5.12. Komponenttitason läpinäkyvyys

Simuloidun läpinäkyvyyden ongelmia voidaan lievittää käyttämällä raskaampaa komponenttitason läpinäkyvyyttä. Komponenttitasolla läpinäkyvä kontrolli piirretään vasta, kun on piirretty muut kontrollit, jotka ovat saman kontrollin sisällä ja piirtojärjestyksessä komponenttitasolla läpinäkyvän kontrollin alapuolella [13]. Tämän takia komponenttitason läpinäkyvyydestä seuraa simuloitua läpinäkyvyyttä parempi vaikutelma läpinäkyvyyden toteutumisesta.

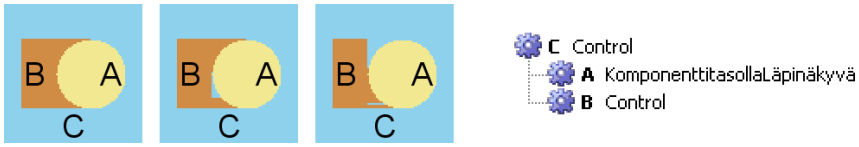
Kokeilun perusteella komponenttitason läpinäkyvyys voidaan asettaa päälle ainakin räätälöityihin kontrolleihin, jotka on periytetty luokasta *UserControl*. Komponenttitason läpinäkyvyys asetetaan päälle ylikirjoittamalla räätälöidyn kontrollin ominaisuus *CreateParams* listauksen 5.1 mukaisesti. Komponenttitason läpinäkyvyyden päälle asettamisessa käytetään Windows API -rajapintaa.

```
protected override CreateParams CreateParams
{
    get
    {
        CreateParams p = base.CreateParams;
        p.ExStyle |= 0x20; //WS_EX_TRANSPARENT
        return p;
    }
}
```

Listaus 5.1. *Komponenttitason läpinäkyvyyden päälle asettaminen*

Kuvassa 5.12 näkyy kuvan 5.11 tilanne muutettuna siten, että räätälöidyssä kontrollissa *A* on asetettu päälle komponenttitason läpinäkyvyys simuloidun sijaan. Kuvan 5.12

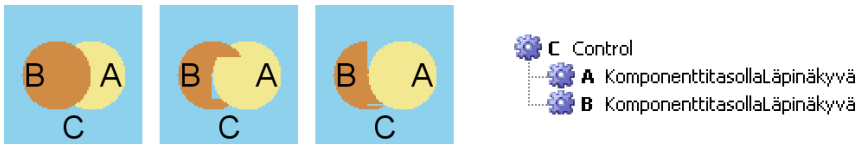
vasemmassa laidassa näkyy hetki, jolloin kokeiluohjelma käynnistyi. Tällöin räätälöidyn kontrollin *A* vasemmista nurkista näkyi oikein valmista kontrollia *B*.



Kuva 5.12. *Räätälöidyn kontrollin komponenttitason läpinäkyvyys toimii kyseenalaisesti*

Kuvassa näkyy keskellä ja oikealla taas tilanne sen jälkeen, kun kokeiluohjelman käyttöliittymäikkunaa liikuteltiin Windowsin työpöydän ulkopuolelle ja takaisin nopeasti. Liikuttelusta seurasi komponenttitason läpinäkyvyyden korjaaman, piirtoalueiden päällekkäisen osan päivittymiseen ongelmia, jollaisia ei tule tavallisesti vastaan .NET-käyttöliittymän piirrossa. Komponenttitason läpinäkyvyys ei siis toimi luotettavasti.

Komponenttitason läpinäkyvyys ei toimi luotettavasti myös silloin, kun kuvan tilanteessa valmis kontrolli *B* vaihdetaan komponenttitasolla läpinäkyvään räätälöityyn kontrolliin, joka piirtää ympyrän samoin kuin räätälöity kontrolli *A*. Näin tehtiin kokeiluohjelmassa, ja kuvan 5.13 vasemmassa laidassa näkyy hetki, jolloin kokeiluohjelma käynnistyi: piirtojärjestyksessä yläpuolella oleva räätälöity kontrolli *A* piirtyikin piirtojärjestyksessä alapuolella olevan räätälöity kontrollin *B* taakse.



Kuva 5.13. *Kahden räätälöidyn kontrollin komponenttitason läpinäkyvyydet toimivat kyseenalaisesti*

Keskellä kuvaa näkyy tilanne sen jälkeen, kun kokeiluohjelman käyttöliittymäikkuna liikutettiin kerran nopeasti Windowsin työpöydän ulkopuolelle ja takaisin: piirtoalueiden päällekkäisen osan päivittyminen oli ongelmallista, mutta räätälöidyt kontrollit *A* ja *B* alkoivat piirtyä oikeaan järjestykseen. Tämän jälkeen, nopeaa liikuttelua jatkettaessa, molemmat piirtyivät oikeaan järjestykseen, mutta piirtoalueiden päällekkäisen osan päivittymisen ongelmat jatkuivat, mikä näkyy kuvan oikeassa laidassa.

6. KOHDISTUS JA NÄPPÄIMISTÖN SYÖTE

Tässä luvussa tutustutaan siihen, miten räätälöity kontrolli pääsee käsittelemään näppäimistön syötettä. Tätä varten esitellään ensin kohdissa 6.1–6.3 kohdistus (focus), joka määrittää, mikä kontrolli kulloinkin saa näppäimistöltä tulevan syötteen (kohdassa 6.1 [3, s. 59]). Kohdistuksen ja sen käyttäytymisen esittelyn jälkeen kohdassa 6.4 pohditaan kohdistusmahdollisuuden poistamista. Kohdassa 6.5 puolestaan keskitytään siihen, miten kohdistus on otettava huomioon räätälöidyssä kontrollissa.

Seuraavaksi tarkastellaan sitä, miten Windows Forms käsittelee näppäimistön syötteen ja ohjaa sen kontrolleihin. Näppäimistön syötteen käsittelystä kuvataan ensin valmiit palvelut kohdassa 6.6 ([5]), ja sitten kohdissa 6.7–6.14 itse käsittely (kohdissa 6.7–6.13 [5]).

6.1. Kohdistettu kontrolli

Kohdistus tarkoittaa valintaa siitä, mikä yksi käyttöliittymäikkuna ja mikä sen kontrollihierarkian kontrolleista on kulloinkin aktiivisena. Windows ohjaa näppäimistön syötteen aktiiviselle kontrollille, josta käytetään nimeä kohdistettu kontrolli. Kohdistus on olemassa, jotta Windows osaa ohjata näppäimistön syötteen käyttäjän valitsemaan kohteeseen.

Kontrolli voi toteuttaa näkyvän kohdistuksen (focus cue), jolloin kontrolli piirtää itsensä korostettuna aina ollessaan kohdistettuna. Esimerkiksi kohdistettu valmis kontrolli *Button* erottuu kohdistamattomasta piirtämällä reunoilleen katkoviivan, joka näkyy kuvan 6.1 vasemmassa laidassa.

Kuva 6.1. *Painike kohdistettuna, painike tavallisena, painike piilotettuna, painike käytöltään estettynä ja painike hiiren alla (mukailtu lähteestä [3, s. 59])*

Samassa kuvassa on myös kohdistamaton painike, piilotettu painike ja painike käytöltään estettynä. Kuvan oikeassa laidassa näkyy hiiren alla oleva painike, joka ilmoittaa hiiren käyttömahdollisuudesta visuaalisella vihjeellä (hot tracking), mikä on eri asia kuin kohdistus.

6.2. Kohdistuksen vaihtaminen

Kohdistus voidaan vaihtaa uuteen kontrolliin Sarkaimella, nuolinäppäimillä, hiirellä tai ohjelmallisesti. Sarkainta käytettäessä kohdistus siirtyy selausjärjestyksessä (tab order)

seuraavaan ja Vaihto-näppäimen kanssa painettuna edelliseen kontrolliin. Selausjärjestys määräytyy kontrollihierarkiasta, kontrollien ominaisuudesta *TabIndex* ja kontrollien z-järjestyksestä.

Etenevä selausjärjestys käy läpi kontrollihierarkian kontrollit välttäen jo läpikäytyjä kontrolleja. Läpikäynti tapahtuu Syvyyteen ensin -haulla, jossa ensisijaisesti aina käydään läpi mahdolliset sisällä olevat kontrollit. Jos seuraavaa kontrollia ei voida valita, selausjärjestys aloittaa alusta.

Saman kontrollin sisällä ja yhdellä kontrollihierarkian tasolla olevien kontrollien keskinäinen selausjärjestys määräytyy niiden ominaisuuden *TabIndex* ja z-järjestyksen perusteella. Ominaisuuden *TabIndex* pienin arvo 0 merkitsee ensimmäistä kontrollia, ja saman arvon sisältävien kontrollien välinen selausjärjestys puolestaan määräytyy z-järjestyksen perusteella. Ominaisuuden *TabIndex* numerointi voidaan siis aloittaa jokaisen kontrollin sisällä aina alusta eli arvosta 0.

Nuolinäppäimet Ylös ja Vasemmalle siirtävät kohdistuksen selausjärjestyksessä edelliseen ja Alas ja Oikealle taas seuraavaan. Nuolinäppäimiä käytettäessä kohdistus ei kuitenkaan siirry kontrollihierarkian uudelle tasolle eli ylöspäin sisältävien kontrollien suuntaan tai alaspäin sisällä olevien kontrollien suuntaan. Lisäksi nuolinäppäimet eroavat Sarkaimesta siten, että kohdistus siirtyy niiden avulla harvemmin. Tämä johtuu siitä, että joidenkin kontrollien yhteydessä nuolinäppäimet on valittu siirtämään kohdistuksen sijaan kontrollin sisäistä kursoria. Esimerkiksi valmiin kontrollin *TextBox* ollessa kohdistettuna nuolinäppäimet siirtävät sen sisäistä tekstinsyöttökursoria.

Hiirtä käytettäessä kohdistus vaihtuu hiiren alla olevaan kontrolliin, kun hiiren ensisijainen painike painetaan alas. Kohdistus vaihdetaan ohjelmallisesti kontrolliin kutsumalla sen parametritonta metodia *Select* tai *Focus*. Niiden välisenä erona jälkimmäinen esittää paluuarvollaan tiedon kohdistuksen vaihtamisen onnistumisesta. Kohdassa 6.3 käsitellään tarkemmin syitä, joiden takia kohdistuksen vaihtaminen voi epäonnistua.

6.3. Kohdistuksen vaihtamisen epäonnistuminen

Kohdistuksen vaihtaminen uuteen kontrolliin eri tavoin on kohdistuksen vaihtopyynnön tekemistä, koska kohdistuksen vaihtaminen voi epäonnistua. Epäonnistuminen voi tapahtua kahdella tavalla: kohdistuksen vaihtaminen estyy, jolloin kohdistettu kontrolli ei vaihdukaan, tai kohdistuksen lyhyeksi aikaa saanut kontrolli vaihtaa kohdistuksen heti sisällään olevalle kontrollille.

Kohdistuksen vaihtaminen voi estyä useista syistä. Nämä syyt jakautuvat korkean ja matalan tason rajoitteisiin. Korkean tason rajoitteet ovat luonteeltaan sellaisia, joiden avulla kontrolli voi vihjata, ettei halua kohdistusta, ja matalan taas sellaisia, joiden takia kontrolli ei voi saada kohdistusta. Korkean tason rajoitteet ovat siis sellaisia, joita vain osa kohdistuksen vaihtotavoista noudattaa, ja matalan sellaisia, joita ei voi olla noudattamatta.

Kohdistuksen vaihtamisen korkean tason rajoitteet ovat kontrollin toive olla saamatta milloinkaan kohdistusta ja toive olla saamatta kohdistusta Sarkaimen avulla. Kontrol-

li ilmaisee ensimmäisestä toiveestaan siten, että kontrollin metodin *GetStyle* kutsu parametrilla *ControlStyles.Selectable* palauttaa epätoden. Jälkimmäisestä toiveestaan kontrolli taas ilmaisee siten, että sen ominaisuus *TabStop* on epätosi.

Kohdistuksen vaihtamisen matalan tason rajoitteet puolestaan ovat käytöltään estetty kontrolli ja piilotettu kontrolli. Nämä rajoitteet ovat olemassa, koska käytöltään estetty kontrolli ei vastaanota näppäimistön syötettä ja ei ole mielekästä, että piilossa oleva kontrolli saa näppäimistön syötteen. Tästä seuraa esimerkiksi, että kohdistuksen saa itse käyttöliittymäikkuna, jos sen sisällä ei ole näkyvissä ja käytettävissä olevia kontrolleja.

Kohdistuksen vaihtamisen matalan tason rajoitteet on koostettu kontrollin matalan tason ominaisuuteen *CanFocus*, joka on tosi, jos matalan tason rajoitteet eivät estä kohdistuksen vaihtamista kyseiseen kontrolliin. Kohdistuksen vaihtamisen matalan tason rajoitteet ja korkean tason rajoite, jossa kontrolli toivoo olla saamatta kohdistusta milloinkaan, on koostettu kontrollin korkean tason ominaisuuteen *CanSelect*.

Valmiit kontrollit toimivat siten, että Sarkaimella, nuolinäppäimillä ja hiirellä tehty kohdistuksen vaihto estyy, jos kontrolli toivoo, ettei saa kohdistusta milloinkaan. Lisäksi valmiissa kontrolleissa Sarkaimella tehty kohdistuksen vaihto estyy, jos kontrolli toivoo, ettei saa kohdistusta Sarkaimen avulla.

Kaikki kontrollit toimivat siten, etteivät korkean tason rajoitteet estä ohjelmallisesti tehtyä kohdistuksen vaihtoa. Metodin *Select* dokumentaatio väittää muuta ja metodin *Focus* dokumentaatio väittää muuta tulkinnanvaraisesti, mutta kokeilujen perusteella sekä metodi *Select* että metodi *Focus* tutkivat kuitenkin vain matalan tason rajoitteita. Näiden metodien välillä ei siis ole toiminnallista eroa, ja ne pakottavat kohdistuksen kontrolliin pelkkien matalan tason rajoitteiden ehdoilla.

Metodeilla *Select* ja *Focus* tehdyissä kokeiluissa periytettiin valmiista kontrollista *Button* räätälöity kontrolli, joka käytti metodejaan *SetStyle* ja *UpdateStyles* ilmoittaakseen toiveestaan, ettei halua milloinkaan kohdistusta. Kokeilut toistettiin myös valmiille kontrolleille *Panel*, *GroupBox*, *PictureBox*, *ProgressBar*, *Splitter* ja *Label*, jotka jo valmiina toivovat samaa asiaa. Kohdistuksen voitiin havaita olevan kontrollissa sillä perusteella, että kontrolli vastaanotti näppäimistön syötteen.

Mikään ei pakota räätälöityä kontrollia ottamaan huomioon kohdistuksen saamiseen liittyviä omia tai muiden kontrollien toiveita, mutta on suositeltavaa seurata valmiiden kontrollien esimerkkiä, ellei ole hyvää perustelua toimia toisin. Esimerkin seuraaminen on suositeltavaa ihan jo yhdenmukaisuudenkin takia. Lisäksi samalla varmistetaan, ettei käyttäjällä ole käytössään kohdistuksen vaihtotapaa, joka ei ota huomioon korkean tason rajoitteita ja voi siten pakottaa kontrollin tilaan, johon kontrollia ei ole suunniteltu.

Kuten tämän kohdan alussa mainittiin, kohdistuksen vaihtaminen voi epäonnistua estymisen lisäksi siten, että kohdistuksen lyhyeksi aikaa saanut kontrolli vaihtaa kohdistuksen heti sisällään olevalle kontrollille. Näin käy, jos kohdistuksen vaihtopyynnön kohteena on säiliökontrolli. Säiliökontrollilla tarkoitetaan valmista kontrollia *ContainerControl* tai sen aliluokkaa. Esimerkiksi valmiit kontrollit *UserControl* ja *Form* ovat säiliökontrolleja (kohta 3.1).

Säiliökontrolli hallinnoi sisällään olevien kontrollien kohdistusta (kohta 6.10). Säiliökontrollin ei ole tarkoitus saada kohdistusta itse, minkä takia säiliökontrolli välittää saamansa kohdistuksen jollekin sisällään olevista kontrolleista.

Kohdistuksen vaihtaminen säiliökontrolliin ei kuitenkaan epäonnistu aina. Jos säiliökontrollin sisällä ei ole kontrollia, jonka ominaisuus *CanSelect* on tosi, säiliökontrolli ei voi välittää kohdistusta eteenpäin ja kohdistus jää säiliökontrolliin.

6.4. Kohdistusmahdollisuuden poistaminen

Kohdistusmahdollisuus voidaan haluta poistaa räätälöidyltä kontrollilta, joka ei sisällä mitään toiminnallisuutta näppäimistön syötteelle eikä hiiren painalluksille. Tällainen räätälöity kontrolli voisi tyypillisesti olla esimerkiksi sellainen, joka ainoastaan piirtää jotakin kuvaa tai tekstiä esittäen jotain tietoa ohjelman nykyisestä tilasta. Jos tällaisella räätälöidyllä kontrollilla olisi kohdistusmahdollisuus, näppäimistön avulla kohdistusta vaihtavasta käyttäjästä tuntuisi turhauttavalta joutua siirtymään kyseisen räätälöidyn kontrollin läpi, koska sen kanssa ei kuitenkaan voi vuorovaikuttaa näppäimistön avulla.

Kohdistusmahdollisuutta ei kuitenkaan kannata poistaa, jos räätälöity kontrolli on sellainen, että sille ei ole toteutettu toiminnallisuutta näppäimistön syötteelle, mutta hiirellä painaminen aiheuttaa jonkin toiminnallisuuden. Tällöin käytettävyyden kannalta on parempi toteuttaa esimerkiksi Välilyönnin tai Rivinvaihdon painamisen yhteyteen sama toiminnallisuus. Näin näppäimistöllä navigoivat käyttäjät voivat käynnistää vastaavan toiminnallisuuden kuin hiirelläkin.

Kohdistusmahdollisuus poistetaan räätälöidyltä kontrollilta Sarkaimen, nuolinäppäinten ja hiiren osalta siten, että räätälöity kontrolli toivoo, ettei saa milloinkaan kohdistusta. Tämä poistaa kohdistusmahdollisuuden riittävissä määrin, koska käyttäjälle ei jää mitään tapaa vaihtaa kohdistusta.

Ohjelmallista kohdistusmahdollisuutta ei sen sijaan voida poistaa räätälöidyltä eikä miltään kontrollilta, joka on näkyvissä ja käytettävissä. Näin on, koska metodit *Select* ja *Focus* tutkivat ainoastaan matalan tason rajoitteita eli kyseisten metodien onnistuminen riippuu ominaisuudesta *CanFocus*. Tilannetta ei voida muuttaa ylikirjoittamisen avulla.

Ohjelmallista kohdistusmahdollisuutta ei myös voida poistaa näkyvissä ja käytettävissä olevalta kontrollilta käyttämällä ratkaisua, joka välittää kontrollin saaman kohdistuksen eteenpäin, joko aina sisältävään tai selausjärjestyksessä seuraavaan kontrolliin (ajatus kohdistusmahdollisuuden poistamisesta välittämällä on peräisin lähteestä [14]). Kumpikaan ratkaisu ei ole kelvollinen, koska molemmista seuraa samoja ongelmia: toisen kontrollin edellyttäminen ja epäintuitiivinen käyttöliittymä.

Toisen kontrollin edellyttäminen tarkoittaa sitä, että kontrollilla, jolta kohdistusmahdollisuus yritetään poistaa kohdistusta välittämällä, on oltava kohteena vähintään yksi toinen kontrolli, jonka ominaisuus *CanSelect* on tosi. Lisäksi, jos kohdistus välitetään eteenpäin aina sisältävään kontrolliin, kontrollin, joka yrittää päästä kohdistuksesta eroon, on oltava kohteena olevan kontrollin sisällä.

Ilman kohteena olevaa toista kontrollia kohdistusta ei voida välittää muualle. Esimerkiksi, jos käyttöliittymäikkunan sisällä on vain yksi kontrolli, joka yrittää välittää saamansa kohdistuksen eteenpäin, kyseinen kontrolli ei pääse eroon kohdistuksesta. Tämän takia kohdistuksen välittämisellä ei voida poistaa kohdistusmahdollisuutta täydellisesti.

Epäintuitiivinen käyttöliittymä tarkoittaa kohdistuksen välittämisen yhteydessä erilaisia ongelmia riippuen siitä, välitetäänkö kohdistus sisältävään vai selausjärjestyksessä seuraavaan kontrolliin. Ensimmäisen vaihtoehdon ongelmana ovat saavuttamattomat kontrollit ja jälkimmäisen ongelmina taas hiiren käyttö sekä selausjärjestyksen toimimattomuus.

Kontrolleja jää saavuttamattomiksi sen takia, että oikea ratkaisu olisi valita sisältävän kontrollin sijaan selausjärjestyksessä seuraava, koska muuten .NET-käyttöliittymään jää kontrolleja, joihin käyttäjä ei voi navigoida näppäimistön avulla. Saavuttamattomat kontrollit ovat esimerkiksi sellaisia, jotka ovat saman kontrollin sisällä kuin kontrolli, joka välittää kohdistuksen sisältävään kontrolliin.

Hiiren käytöstä taas tulee käyttäjän kannalta epäintuitiivista, koska kohdistus ei vaihdukaan hiiren alla olevaan kontrolliin, kun hiiren ensisijainen painike painetaan alas. Sen sijaan kohdistus vaihtuu alla olevan kontrollin viereiseen kontrolliin.

Käyttäjän kannalta on epäintuitiivista myös, että edellisten kontrollien suuntaan etenevä selausjärjestys lakkaa toimimasta. Tämä johtuu siitä, että kontrolli välittää kohdistuksen heti toiseen suuntaan. Tällöin käyttäjä ei pääse edellisten kontrollien suuntaan etenevässä selausjärjestyksessä kyseisen kontrollin ohi. Toiseen suuntaan eteneminen kuitenkin yhä toimisi, koska kontrolli välittää kohdistuksen aina suuntaan, joka on oikea seuraavien kontrollien suuntaan etenevän selausjärjestyksen kannalta.

Selausjärjestyksen korjaamiseksi pitäisi selvittää kohdistuksen vaihtamiseen käytetyt näppäimet. Näin kohdistuksen välittävä kontrolli tietäisi, mikä on oikea suunta välittää kohdistus. Selausjärjestyksen korjaaminen on kuitenkin vaivalloista verrattuna seuraavaksi esiteltävään ratkaisuun, jolla kohdistusmahdollisuus poistetaan täydellisesti.

Näkyvillä olevalta räätälöidyltä kontrollilta poistetaan kohdistusmahdollisuus täydellisesti estämällä räätälöidyn kontrollin käyttö. Muuta tapaa ei ole olemassa. Käytön estämisen jälkeen räätälöity kontrolli ei voi vastaanottaa näppäimistön syötettä eikä hiiren painalluksia. Käytön estämisen jälkeen esimerkiksi räätälöity kontrolli, joka ainoastaan piirtää jotakin esittäen jotain tietoa ohjelman nykyisestä tilasta, toimii tarkoituksenmukaisesti. Jos käyttöä ei voida estää, koska hiiren painalluksia halutaankin käsitellä, ratkaisuna ei ole kohdistusmahdollisuuden poistaminen vaan tuen toteuttaminen vastaavaa näppäimistöllä tapahtuvaa käyttöä varten, kuten aiemmin todettiin.

Lisähuomiona, näkyvillä ja käytettävissä oleva kontrolli ei voi suojautua mitenkään, jos sille pakotetaan kohdistus ohjelmallisesti metodinsa *Select* tai *Focus* avulla. Yleisesti on suositeltavaa kunnioittaa kontrollin toivetta olla saamatta kohdistusta milloinkaan. Tämän takia onkin hyvän tavan mukaista kutsua kontrollin metodia *Select* tai *Focus* vain, jos kohteen ominaisuus *CanSelect* on tosi.

6.5. Näkyvän kohdistuksen ja hiirituen toteuttaminen

Kohdistus on tavallisesti otettava huomioon räätälöidyssä kontrollissa siten, että se toteuttaa ulkoasuunsa näkyvän kohdistuksen ja toteuttaa tuen kohdistuksen vaihtamiseen hiiren avulla. Näkyvä kohdistus mahdollistaa sen, että käyttäjä voi tunnistaa kohdistetun kontrollin ja siten tietää, mikä kontrolli saa näppäimistöltä tulevan syötteen. Hiirituen toteuttaminen taas tekee räätälöidystä kontrollista sellaisen, että siihen vaihtuu kohdistus näppäimistön lisäksi hiirenkin avulla, mikä on käyttäjälle intuitiivista.

Räätälöity kontrolli toteuttaa näkyvän kohdistuksen piirtämällä itsensä tilansa mukaisesti. Räätälöity kontrolli voi seurata kohdistustaan kuvaavan ominaisuutensa *Focused* muuttumista tapahtumillaan *GotFocus* ja *LostFocus*.

Jotta kohdistus voidaan vaihtaa räätälöityyn kontrolliin Sarkaimen ja nuolinäppäinten lisäksi hiirelläkin, räätälöidyn kontrollin on seurattava tapahtumaansa *MouseDown*. Kyseisen tapahtuman yhteyteen kirjoitettava toiminnallisuus riippuu räätälöidyn kontrollin kantaluokasta. Räätälöity kontrolli voi nimittäin olla säiliökontrolli, jollaisten on tarkoitus välittää saamansa kohdistus jollekin sisällään olevista kontrolleista kohdistuksen vastaanottamisen sijaan.

Jos räätälöity kontrolli on periytetty suoraan valmiista kontrollista *Control* tai *ScrollableControl* (kohta 3.1), räätälöidyn kontrollin on kutsuttava kyseisen tapahtuman yhteydessä metodiaan *Select* tai *Focus*. Sen sijaan suoraan valmiista kontrollista *ContainerControl* periytetyn räätälöidyn kontrollin on kutsuttava metodinsa *Select* parametrillista versiota kahdella arvoltaan tosi olevalla parametrilla, jotta kohdistus välittyy oikein painetun räätälöidyn kontrollin sisällä olevaan kontrolliin. Valmiista kontrollista *UserControl* periytetyn räätälöidyn kontrollin puolestaan riittää kohdistuksen välittämiseksi kutsua metodiaan *Select* tai *Focus*, vaikka kyseessä onkin säiliökontrolli.

Kuten aiemmin todettiin, metodia *Select* tai *Focus* on hyvä kutsua vain, jos kohteen eli tässä tapauksessa räätälöidyn kontrollin ominaisuus *CanSelect* on tosi. Näin tuetaan samalla yhdenmukaista toimintaa, jossa silloin, kun räätälöity kontrolli ei voi saada kohdistusta Sarkaimella ja nuolinäppäimillä, räätälöity kontrolli ei saa kohdistusta hiirelläkään.

6.6. Näppäimistön syötteen käsittelyn valmiit palvelut

Windows Forms tarjoaa .NET-käyttöliittymille palveluita valmiina liittyen näppäimistön syötteen käsittelyyn. Valmiit palvelut tarjotaan siten, että räätälöidyt kontrollit pääsevät muuttamaan näppäimistön syötteen käsittelyn yksityiskohtia perinpohjaisesti. Valmiiden palveluiden esittelyä varten kuvataan ensin näppäimistön toimintaa.

Näppäimistön käytöstä seuraa näppäinpainalluksia. Näppäinpainallus on yhden näppäimen painalluskerta, joka koostuu vaiheista näppäin painuu alas, näppäin on hetken pohjassa ja näppäin vapautuu ylös. Näppäinpainalluksen eri vaiheiden aikana muut näppäimet voivat käydä läpi omia näppäinpainallustensa vaiheita.

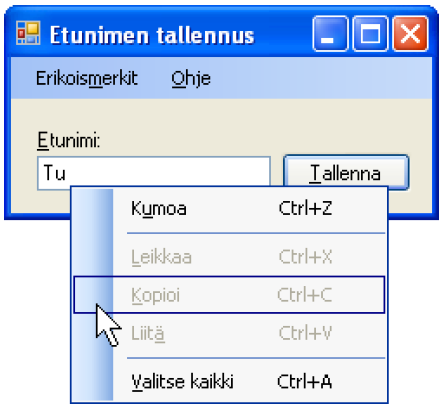
Windows Forms tarjoaa .NET-käyttöliittymille valmiina palveluina tuen erikoisnäppäinten näppäinpainallusten käsittelyyn sekä mahdollisuuden käsitellä tavallisten näppäinten näppäinpainalluksia. Erikoisnäppäimet jaotellaan komento-, navigointi- ja muistisääntönäppäimiin.

Komentonäppäimet ovat erikoisnäppäimiä, jotka suorittavat .NET-käyttöliittymän käyttäjälleen tarjoaman komennon. Komentonäppäimiä ovat esimerkiksi asiayhteysvalikon ja käyttöliittymäikkunan valikon tarjoamat pikanäppäimet. Esimerkiksi näppäin S voi olla komentonäppäin näppäimen Ctrl kanssa painettuna.

Navigointinäppäimet taas ovat erikoisnäppäimiä, joilla käyttäjä voi navigoida .NET-käyttöliittymässä. Navigointinäppäimiä ovat esimerkiksi nuolinäppäimet sekä näppäimet Esc, Rivinvaihto ja Sarkain.

Muistisääntönäppäimet (access key, mnemonic) puolestaan ovat erikoisnäppäimiä, jotka aiheuttavat pikatoiminnon yksinään tai näppäimen Alt kanssa painettuna. Tällainen pikatoiminto kohdistuu siihen .NET-käyttöliittymän osaan, jossa on tavallisesti alleviivaamalla korostettu yksi kirjain, ja tätä kirjainta vastaava näppäin on muistisääntönäppäin. Tavallisesti muistisääntönäppäimiä käytetään esimerkiksi erilaisten valikoiden eri vaihtoehdoissa ja .NET-käyttöliittymän painikkeissa.

Kuvassa 6.2 on esimerkkejä erilaisista erikoisnäppäimistä. Esimerkiksi näppäin Z on komentonäppäin näppäimen Ctrl kanssa painettuna. Näppäin E taas on muistisääntönäppäin näppäimen Alt kanssa painettuna. Tällöin pikatoiminto kohdistuu valmiiseen kontrolliin *Label*, joka näyttää tekstiä Etunimi:. Valmiiseen kontrolliin *Label* liittyvä pikatoiminto on kohdistuksen vaihtaminen itsestään selausjärjestyksessä seuraavaan eli tässä tilanteessa tekstikenttään. Muistisääntönäppäimen T ja näppäimen Alt avulla taas voidaan painaa kuvan painiketta. Muistisääntönäppäin V yksinään painettuna puolestaan valitsee asiayhteysvalikon alimman vaihtoehdon.



Kuva 6.2. *Esimerkkejä komento- ja muistisääntönäppäimistä*

Erikoisnäppäinten valmis tuki ottaa huomioon sen, että ne edellyttävät toiminnallisuutta, joka mahdollistaa niiden näppäinpainallusten käsittelyn muissakin kontrolleissa kuin kohdistetussa. Esimerkiksi tallentamisen mahdollistavan komentonäppäimen on tavallisesti toimittava kohdistetusta kontrollista riippumatta. Samoin navigointinäppäinten näppäinpainallusten käsittely tapahtuu tavallisesti säiliökontrollissa, jonka sisällä koh-

distettu kontrolli on. Muistisääntönäppäinten on taas voitava kohdistaa pikatoiminto .NET-käyttöliittymän osaan kulloinkin kohdistetusta kontrollista riippumatta.

Valmiina tarjottu mahdollisuus käsitellä tavallisten näppäinten näppäinpainalluksia puolestaan ottaa huomioon sen, että mukavuussyistä näppäinpainallukset saatetaan haluta käsitellä käyttöliittymäikkunassa. Tämän ansiosta esimerkiksi aina samoin toimivien pikanäppäinten näppäinpainallusten käsittely voidaan toteuttaa käyttöliittymäikkunaan sen sijaan, että sama käsittely on työläästi toteutettava erikseen jokaiseen kohdistuksen mahdollisesti saavaan kontrolliin.

Mahdollisuus käsitellä tavallisten näppäinten näppäinpainalluksia käyttöliittymäikkunassa tarjotaan siten, oletuksena ne tarjotaan vain kohdistetulle kontrollille. Jos käyttöliittymäikkuna valitsee toisin, tavallisten näppäinten näppäinpainallukset tarjotaan ensin käyttöliittymäikkunalle ja vasta sitten kohdistetulle kontrollille.

6.7. Näppäinpainalluksen kuluttaminen

Windows Forms tarjoaa näppäinpainalluksen käsiteltäväksi erilaisille metodeille niitä kutsumalla. Ne voivat olla kohdistetun kontrollin, jonkun muun kontrollin tai jonkin muun luokan metodeja. Jokainen metodi voi halutessaan suorittaa jotain toiminnallisuutta näppäinpainallukseen liittyen. Riippumatta suoritetusta tai suorittamatta jätetystä toiminnallisuudesta, jokaisen metodin on lopuksi päätettävä, kuluttaako metodi näppäinpainalluksen.

Jos metodi kuluttaa näppäinpainalluksen, Windows Forms ei tarjoa näppäinpainallusta käsiteltäväksi enää seuraaville metodeille. Näin varmistetaan, että näppäinpainallus aiheuttaa enintään yhden toiminnallisuuden. Tätä ratkaisua voidaan kiertää päättämällä jättää näppäinpainallus kuluttamatta, vaikka jokin toiminnallisuus suoritetaankin. Yleisesti on kuitenkin käyttäjän kannalta intuitiivisempaa, ettei näppäinpainallus aiheuta useita toiminnallisuuksia.

Metodi, jolle näppäinpainallus tarjotaan käsiteltäväksi, saa tietoa näppäinpainalluksesta parametrinsa välityksellä. Metodi ilmoittaa päätöksestään näppäinpainalluksen kuluttamisen suhteen paluuarvollaan.

6.8. Näppäinpainallusten käsittelyn vaiheet

Windows Forms tarjoaa näppäimistön syötteen käsittelyyn liittyvät valmiit palvelut siten, että näppäimistön syötteen käsittely sisältää erilaisia vaiheita, jotka yksittäinen näppäinpainallus käy läpi. Seuraavaksi käydään läpi näitä vaiheita. Niiden yksityiskohtien tunteminen mahdollistaa näppäinpainallusten käsittelyn perinpohjaisen muuttamisen, kuten esimerkiksi uusien erikoisnäppäinten toteuttamisen räätälöidyille kontrolleille valmiita palveluita hyödyntäen.

Näppäinpainallusten käsittelyn vaiheiden yksityiskohtien tunteminen ei ole tarpeen, jos tyytyy tavallisten näppäinten näppäinpainallusten käsittelyyn, jota esitellään tarkemmin kohdissa 6.13–6.14. Yksityiskohdat tuntemalla kohdistetulta räätälöidyltä kontrol-

lilta voidaan kuitenkin vaivattomasti poistaa erikoisnäppäinten näppäinpainallusten käsittely. Tällöin se ei pääse kuluttamaan näppäinpainalluksia, jolloin kaikki näppäinpainallukset varmasti tarjotaan räätälöidyn kontrollin tavallisten näppäinten näppäinpainallusten käsittelyyn. Poistamista käsitellään tarkemmin kohdassa 6.10.

Näppäinpainallusten käsittelyn vaiheet lähtevät liikkeelle siitä, kun näppäinpainallusten seurauksena Windows API -rajapinnan läpi tulee Windowsin viestejä (Windows messages), jotka Windows Forms käsittelee. Windowsin viestejä käytetään myös muihin tarkoituksiin kuin näppäinpainalluksiin liittyvän tiedon välittämiseen, joten kaikki myöhemmin mainittavat metodit eivät liity ainoastaan näppäinpainallusten käsittelyyn. Windows Forms käyttää käsittelynsä aikana tietuetta *System.Windows.Forms.Message* esittämään Windowsin viestiä. Käsittelynsä aikana Windows Forms myös jalostaa raa'at Windowsin viestit korkeamman tason tietomuotoon.

Windows Forms suorittaa käsittelyn siten, että Windows Forms tarjoaa näppäinpainallukset käsiteltäviksi erilaisille metodeille, jotka vastaavat näppäinpainallusten käsittelyn vaiheita. Jokainen vaihe voi siis kuluttaa näppäinpainalluksen, jolloin käsittely pysähtyy (kohta 6.7). Monet näistä metodeista ovat ylikirjoitettavia, jotta aliluokat voivat muokata näppäinpainallusten käsittelyä perinpohjaisesti. Ylikirjoittavan metodin on tarkoitus kuluttaa näppäinpainallus ilmoittaen siitä paluuarvollaan tai palauttaa kantaluokkansa toteutuksen paluuarvo.

Näppäinpainallusten käsittelyn vaiheet ovat suoritusjärjestyksessä esisuodatusvaihe, esikäsittelyvaihe, käsittelyvaihe ja oletuskäsittelyvaihe. Seuraavaksi nämä esitellään lyhyesti. Niitä käsitellään tarkemmin tämän luvun kohdissa 6.9–6.12.

Esisuodatusvaihe mahdollistaa näppäinpainallusten kuluttamisen koko ohjelman tasolla riippumatta siitä, mikä kontrolli on kohdistettuna. Esikäsittelyvaiheessa tapahtuu erikoisnäppäinten näppäinpainallusten käsittely. Ensin käsitellään komentonäppäinten, sitten navigointinäppäinten ja lopuksi muistisääntönäppäinten näppäinpainallukset. Käsittelyvaiheessa tapahtuu tavallisten näppäinten näppäinpainallusten käsittely. Käsittelyvaiheessa näppäinpainallus tarjotaan kulutettavaksi käyttöliittymäikkunalle ennen kohdistettua kontrollia, jos käyttöliittymäikkuna on valinnut niin. Oletuskäsittelyvaihe tarjoaa Windowsin oletuskäsittelyn, joka mahdollistaa joitakin perustoimintoja.

Koska näppäinpainallus koostuu vaiheista näppäin painuu alas, näppäin on hetken pohjassa ja näppäin vapautuu ylös, yksi näppäinpainallus käy läpi näppäinpainallusten käsittelyn vaiheet vähintään kolmesti. Näin näppäinpainallusten käsittelyn vaihe voi toimia eri tavoin riippuen näppäinpainalluksen vaiheesta. Esimerkiksi esikäsittelyvaiheeseen on toteutettu komento- ja navigointinäppäinten näppäinpainallusten käsittely siihen näppäinpainalluksen vaiheeseen, jossa näppäin painuu alas, ja muistisääntönäppäinten näppäinpainallusten käsittely taas siihen näppäinpainalluksen vaiheeseen, jossa näppäin on hetken pohjassa.

Näppäinpainalluksen kuluttaminen tarkoittaa siis täsmällisesti sanottuna näppäinpainalluksen jonkun vaiheen kuluttamista. Näppäinpainalluksen vaiheen kuluttamisen seurauksena näppäinpainalluksen kyseisen vaiheen käsittely pysähtyy, mutta jäljelle jääneet näppäinpainalluksen vaiheet tarjotaan yhä kulutettaviksi.

6.9. Näppäinpainalluksen esisuodatusvaihe

Näppäinpainalluksen esisuodatusvaiheessa Windows Forms kutsuu metodeja ohjelman tasolla siinä missä muissa vaiheissa Windows Forms kutsuu kohdistetun kontrollin metodeja. Esisuodatusvaiheessa kohdistetulla kontrollilla ei siis ole merkitystä.

Esisuodatusvaiheessa Windows Forms kutsuu metodia *PreFilterMessage* niille olioille, jotka on lisätty suodatinjonoon. Olio voidaan lisätä suodatinjonoon, jos olio toteuttaa rajapinnan *System.Windows.Forms.IMessageFilter*. Tällainen olio lisätään suodatinjonoon kutsumalla luokan *System.Windows.Forms.Application* metodia *AddMessageFilter*.

Suodatinjonon oliot saavat siis mahdollisuuden näppäinpainalluksen kuluttamiseen ennen käyttöliittymäikkunaa ja sen sisällä olevia kontrolleja. Jos näppäinpainallusta ei kulutettu esisuodatusvaiheessa, näppäinpainallus siirtyy esikäsittelyvaiheeseen.

6.10. Näppäinpainalluksen esikäsittelyvaihe

Näppäinpainalluksen esikäsittelyvaiheessa Windows Forms kutsuu kohdistetun kontrollin metodia *PreProcessMessage*. Se aloittaa kontrollien metodien kutsuketjuja, jotka muun muassa etenevät pois kohdistetusta kontrollista jatkaen rekursiivisesti aina sisältävään kontrolliin saavuttaen lopulta käyttöliittymäikkunan. Kuten yleensäkin, näppäinpainalluksen kuluttaminen keskeyttää kutsuketjun.

Metodi *PreProcessMessage* kutsuu kohdistetun kontrollin eri metodeja riippuen näppäinpainalluksen vaiheesta, kuten aiemmin mainittiin. Jos näppäin painuu alas, metodit ovat kutsujärjestyksessä kontrollin metodit *ProcessCmdKey*, *IsInputKey* ja *ProcessDialogKey*. Jos taas näppäin on hetken pohjassa, metodit ovat kutsujärjestyksessä kontrollin metodit *IsInputChar* ja *ProcessDialogChar*.

Metodiin *ProcessCmdKey* on toteutettu komentonäppäinten näppäinpainallusten käsittely. Se antaa näppäinpainalluksen kulutettavaksi ensin samaan kontrolliin liittyvälle asiayhteysvalikolle ja sitten saman kontrollin sisältävälle kontrollille kutsuen sen samaa metodia. Kuluttamattoman näppäinpainalluksen päädyttyä lopulta käyttöliittymäikkunaan se antaa näppäinpainalluksen kulutettavaksi valikolleen.

Räätälöity kontrolli voi metodin *ProcessCmdKey* avulla toteuttaa vapaavalintaiselle kontrollihierarkian tasolle toiminnallisuuden, joka kuluttaa näppäinpainalluksia sisällään olevalta kohdistetulta kontrollilta. Saman metodin avulla voidaan myös poistaa komentonäppäinten näppäinpainallusten käsittely.

Metodi *IsInputKey* on poikkeava metodi, koska se ei voi kuluttaa näppäinpainallusta. Sen sijaan kyseinen metodi voi paluuarvollaan ohittaa näppäinpainalluksen päätymisen navigointinäppäinten näppäinpainallusten käsittelyyn. Räätälöidyn kontrollin on ylikirjoitettava tämä metodi esimerkiksi silloin, kun räätälöity kontrolli haluaa kuluttaa Sarkaimen näppäinpainallukset. Jos käsittely ohitetaan, seuraava metodi *ProcessDialogKey* jätetään kutsumatta.

Metodin *IsInputKey* tavoin, näppäinpainalluksen päätyminen navigointinäppäinten näppäinpainallusten käsittelyyn voidaan ohittaa myös seuraamalla kontrollin tapahtumaa *PreviewKeyDown* ja asettamalla sen käsittelijän saaman parametrin ominaisuus *IsInputKey* todeksi. Tapahtuma *PreviewKeyDown* nostetaan metodien *ProcessCmdKey* ja *ProcessDialogKey* kutsumisen välissä.

Metodiin *ProcessDialogKey* on toteutettu navigointinäppäinten näppäinpainallusten käsittely. Se antaa näppäinpainalluksen kulutettavaksi sisältävälle kontrollille kutsuen sen samaa metodia. Näppäinpainallus päätyy lopulta säiliökontrolliin, johon on toteutettu käsittely, joka voi kuluttaa näppäinpainalluksen. Kuluttamaton näppäinpainallus päätyy lopulta käyttöliittymäikkunaan (joka sekin on säiliökontrolli).

Metodi *IsInputChar* on myös poikkeava metodi, koska sekään ei voi kuluttaa näppäinpainallusta. Sen sijaan kyseinen metodi voi paluuarvollaan ohittaa näppäinpainalluksen päätymisen muistisääntönäppäinten näppäinpainallusten käsittelyyn. Jos käsittely ohitetaan, seuraava metodi *ProcessDialogChar* jätetään kutsumatta.

Metodiin *ProcessDialogChar* on toteutettu muistisääntönäppäinten näppäinpainallusten käsittely. Se antaa näppäinpainalluksen kulutettavaksi sisältävälle kontrollille kutsuen sen samaa metodia. Näppäinpainallus päätyy lopulta säiliökontrolliin. Sen kyseinen metodi kutsuu saman kontrollin ja jokaisen sen sisällä olevan kontrollin metodia *ProcessMnemonic*, johon on toteutettu käsittely, joka voi kuluttaa näppäinpainalluksen. Kuluttamaton näppäinpainallus päätyy lopulta käyttöliittymäikkunaan (joka sekin on säiliökontrolli).

Esikäsittelyvaiheen edellä kuvattujen yksityiskohtien ansiosta erikoisnäppäimet voivat toimia, vaikkeivät ne liittyisikään kohdistettuun kontrolliin (ks. kohta 6.6). Erikoisnäppäinten käsittely on siis toteutettu valmiisiin kontrolleihin, kuten luokkiin *Control*, *ContainerControl* ja *Form*. Saman metodin kutsuminen sisältävälle kontrollille on toteutettu joidenkin edellä kuvattujen metodien kohdalla luokkaan *Control*. Sellaisen metodin ylikirjoittavan räätälöidyn kontrollin on kuluttamattoman näppäinpainalluksen yhteydessä kutsuttava kantaluokkansa toteutusta, kuten aiemmin mainittiin. Muutoin näppäinpainallus ei ohjaudu sisältäviin kontrolleihin ja erikoisnäppäimet eivät toimi.

Esikäsittelyvaihe sisältää monia metodeja, jotka räätälöity kontrolli voi ylikirjoittaa jotain tavoiteltua toiminnallisuutta varten. Ylikirjoitettavan metodin valinta voi tuntua vaikealta, koska saman toiminnallisuuden voi saavuttaa monen metodin avulla. Lähteessä [5] on esimerkkejä tämän valinnan tekemisestä erilaisissa tilanteissa. Valinta onkin tilannekohtaista, mutta tavallisesti riittävät metodit *IsInputKey* ja *IsInputChar* sekä näppäinpainalluksen kuluttaminen kohdistetussa kontrollissa vasta näppäinpainalluksen käsittelyvaiheessa.

Seuraava näppäinpainallusten käsittelyn vaihe onkin käsittelyvaihe, jossa näppäinpainallus päätyy tavallisten näppäinten näppäinpainallusten käsittelyyn. Jos näppäinpainallusta siis ei kulutettu esikäsittelyvaiheessa erikoisnäppäimenä, näppäinpainallus siirtyy käsittelyvaiheeseen.

6.11. Näppäinpainalluksen käsittelyvaihe

Näppäinpainalluksen käsittelyvaiheessa Windows Forms kutsuu kohdistetun kontrollin metodia *WndProc*. Se aloittaa kontrollien metodien kutsuketjun, joka etenee pois kohdistetusta kontrollista jatkaen rekursiivisesti aina sisältävään kontrolliin saavuttaen lopulta käyttöliittymäikkunan. Tämä kutsuketju mahdollistaa toiminnallisuuden, jolla näppäinpainallus voidaan, käyttöliittymäikkunan valinnan mukaisesti, tarjota kulutettavaksi käyttöliittymäikkunalle ennen kohdistettua kontrollia (ks. kohta 6.6). Näppäinpainallusta ei pidä kuluttaa ennen käyttöliittymäikkunan saavuttamista, koska muuten käyttöliittymäikkuna ei saa mahdollisuutta kuluttaa näppäinpainallusta ensimmäisenä.

Käsittelyvaihe tarjoaa näppäinpainalluksen kulutettavaksi näppäimistötapahtumien kautta. Ne ovat kontrollin tapahtumat *KeyDown*, *KeyPress* ja *KeyUp*, jotka vastaavat näppäinpainalluksen vaiheita näppäin painuu alas, näppäin on hetken pohjassa ja näppäin vapautuu ylös. Näppäinpainallus on matalan tason käsite verrattuna näppäimistötapahtumaan. Esimerkkinä tästä näppäimistötapahtuman parametrina on Windowsin viestin sijaan sen oleellinen sisältö jalostetussa muodossa. Näppäimistötapahtumia käsitellään tarkemmin kohdassa 6.13.

Käsittelyvaihe nostaa näppäimistötapahtuman joko ainoastaan kohdistetulle kontrollille tai ensin käyttöliittymäikkunalle ja sitten kohdistetulle kontrollille. Käyttöliittymäikkuna valitsee näistä tavoista jommankumman ominaisuutensa *KeyPreview* avulla. Se määrittää oletuksena vaihtoehdoista ensimmäisen, kuten mainittiin kohdassa 6.6.

Näppäimistötapahtuma kulutetaan asettamalla sen parametrin ominaisuus *Handled* todeksi. Näppäimistötapahtuman kuluttamisesta seuraa, että näppäinpainallus tulee kulutetuksi, jolloin käsittely päättyy, kuten yleensäkin. Jos käyttöliittymäikkunalle nostettu näppäimistötapahtuma kulutetaan, sitä ei nosteta enää kohdistetulle kontrollille.

Näppäimistötapahtuman nostamiseksi metodi *WndProc* kutsuu kohdistetun kontrollin metodia *ProcessKeyMessage*. Tämä puolestaan kutsuu sisältävän kontrollin metodia *ProcessKeyPreview*. Se antaa kuluttamattoman näppäinpainalluksen edelleen kulutettavaksi sisältävälle kontrollille kutsuen sen samaa metodia. Näin näppäinpainallus päätyy lopulta käyttöliittymäikkunaan. Jos sen ominaisuus *KeyPreview* on tosi, käyttöliittymäikkuna kutsuu metodiaan *ProcessKeyEventArgs*. Seuraavaksi kutsuketju palautuu takaisin kohdistettuun kontrolliin, jonka metodi *ProcessKeyMessage* kutsuu kuluttamattomalle näppäinpainallukselle kohdistetun kontrollin metodia *ProcessKeyEventArgs*.

Metodi *ProcessKeyEventArgs* toimii sekä käyttöliittymäikkunassa että kohdistetussa kontrollissa samoin eli nostaa näppäimistötapahtuman. Tämä tapahtuu siten, että, riippuen Windowsin viestin kuvaamasta näppäinpainalluksen vaiheesta, kyseinen metodi kutsuu kyseisen kontrollin metodia *OnKeyDown*, *OnKeyPress* tai *OnKeyUp*. Jos näppäimistötapahtuma kulutetaan, metodi *ProcessKeyEventArgs* esittää paluuarvollaan, että näppäinpainalluskin kulutettiin.

Käsittelyvaiheen toiminnallisuus on toteutettu valmiisiin kontrolleihin *Control* ja *Form*. Räätälöidyn kontrollin ei pidä ylikirjoittaa näiden kontrollien metodeja *WndProc*, *ProcessKeyMessage*, *ProcessKeyPreview* ja *ProcessKeyEventArgs* kuluttaakseen näp-

päinpainalluksen tavallisten näppäinten näppäinpainallusten käsittelyssä, vaan räätälöity kontrolli tavallisesti seuraa näppäimistötapahtumiaan ja kuluttaa niitä.

Jos mikään metodin *WndProc* käynnistämän kutsuketjun metodeista ei kuluttanut näppäinpainallusta, kyseinen metodi kutsuu lopuksi edelleen kontrollin metodia, joka vastaa näppäinpainalluksen oletuskäsittelyvaiheesta. Koska metodi *WndProc* kutsuu suoraan oletuskäsittelyvaiheen metodia, metodilla *WndProc* ei ole paluuarvoa. Metodi *WndProc* ei siis raportoi näppäinpainalluksen kuluttamisesta millekään osapuolelle.

6.12. Näppäinpainalluksen oletuskäsittelyvaihe

Näppäinpainalluksen oletuskäsittelyvaiheessa kohdistetun kontrollin metodin *WndProc* toteutus luokassa *Control* kutsuu saman kontrollin metodia *DefWndProc*. Se toteuttaa Windowsin oletuskäsittelyn näppäinpainallukselle, jota ei ole kulutettu missään aiemmassa näppäinpainallusten käsittelyn vaiheessa.

Jos Windowsin oletuskäsittely ei kuluta näppäinpainallusta, viimeinen näppäinpainallusten käsittelyn vaihe on saanut mahdollisuuden näppäinpainalluksen kuluttamiseen. Tämän jälkeen ei ole olemassa seuraavaa vaihetta, joka voisi kuluttaa yhä kuluttamattomaksi jääneen näppäinpainalluksen, joten metodilla *DefWndProc* ei ole paluuarvoa, joka esittäisi tiedon näppäinpainalluksen kuluttamisesta. On siis yhdentekevää, päättääkö kyseinen metodi kuluttaa näppäinpainalluksen vai ei, koska metodi ei voi esittää päätöstään eteenpäin.

6.13. Näppäimistötapahtumat

Windows Forms nostaa kontrollin näppäimistötapahtumia *KeyDown*, *KeyPress* ja *KeyUp*. Näppäimistötapahtumat nostetaan vain kohdistetulle kontrollille tai ensin käyttöliittymäikkunalle ja sitten kohdistetulle kontrollille, mikä riippuu käyttöliittymäikkunan ominaisuudesta *KeyPreview*, kuten aiemmin mainittiin.

Näppäimistötapahtumia seuraamalla kontrolli voi vastaanottaa ja käsitellä näppäimistön syötettä niiden näppäinpainallusten osalta, joita ei kulutettu erikoisnäppäiminä. Kuten aiemmin kohdassa 6.10 mainittiin, räätälöity kontrolli voi ohittaa navigointinäppäinten näppäinpainallusten käsittelyn ylikirjoittamalla kontrollin metodin *IsInputKey* ja muistisääntönäppäinten näppäinpainallusten käsittelyn ylikirjoittamalla kontrollin metodin *IsInputChar*. Ohittamisen jälkeen Windows Forms nostaa tapahtuman *KeyDown* myös aiemmin navigointinäppäiminä toimineille näppäimille ja samoin tapahtuman *KeyPress* myös aiemmin muistisääntönäppäiminä toimineille näppäimille.

Näppäimistötapahtumat nostetaan seuraavassa järjestyksessä jokaista näppäinpainallusta kohti. Kun näppäin painuu alas, nostetaan tapahtuma *KeyDown*. Tämän jälkeen nostetaan tapahtuma *KeyPress*. Kun näppäin vapautuu ylös, nostetaan tapahtuma *KeyUp*. Jos taas näppäintä pidetään pitkään pohjassa, tapahtumia *KeyDown* ja *KeyPress* nostetaan useasti vuoron perään, mutta näppäimen lopulta vapautuessa ylös tapahtuma *KeyUp* nostetaan vain kerran.

Näistä tapahtuma *KeyPress* nostetaan vain, jos näppäinpainallus tai samanaikaiset näppäinpainallukset tuottavat kirjaimen. Tämän vuoksi tapahtumalla *KeyPress* ei voida käsitellä näppäinpainalluksia esimerkiksi nuolinäppäimiltä, funktionäppäimiltä eikä näppäimiltä Sarkain, Lisää, Poista, Alkuun, Loppuun, Sivu ylös, Sivu alas ja Alt. Tapahtuman *KeyPress* tyyppiä *KeyPressEventArgs* olevan parametrin ominaisuus *KeyChar* sisältää kirjaimen, jonka näppäinpainallus tai samanaikaiset näppäinpainallukset tuottivat. Kirjain on iso tai pieni riippuen siitä, oliko Vaihto samaan aikaan pohjassa ja oliko isojen kirjainten lukko päällä.

KeyDown ja *KeyUp* ovat tapahtumia, jotka nostetaan aina jokaisen näppäimen alas painumisen ja ylös vapautumisen yhteydessä. Tapahtumien *KeyDown* ja *KeyUp* tyyppiä *KeyEventArgs* olevan parametrin ominaisuus *KeyCode* sisältää tiedon varsinaisesta alas painuneesta tai ylös vapautuneesta näppäimestä. Parametrin ominaisuus *Modifiers* puolestaan kuvaa mahdollisesti samaan aikaan alas painuneiden muokkaavien näppäinten yhdistelmän. Muokkaavia näppäimiä ovat Vaihto, Ctrl ja Alt. Parametrin ominaisuus *KeyData* taas sisältää koosteen ominaisuuksien *KeyCode* ja *Modifiers* arvoista.

Tapahtuman *KeyDown* parametrilla on ominaisuus *SuppressKeyPress*, jonka todeksi asettaminen estää alas painuneeseen näppäimeen liittyvän tapahtuman *KeyPress* nostamisen. Tällöin kuitenkin yhä nostetaan samaan näppäimeen liittyvä tapahtuma *KeyUp*. Ominaisuuden *SuppressKeyPress* avulla voidaan esimerkiksi estää muiden merkkien kuin numeroiden ilmestyminen tekstikenttään [3, s. 61–62].

Lopuksi on yhteenvetona esimerkki näppäimistötapahtumien noston järjestyksestä. Tilanteessa, jossa ensin Vaihto painuu alas ja sitten näppäin A, nostetaan tapahtuma *KeyDown* ensin Vaihdolle ja sitten A:lle. Tämän jälkeen A:lle nostetaan tapahtuma *KeyPress*, joka sisältää ison a-kirjaimen. Kun heti perään ensin A vapautuu ylös ja sitten Vaihto, nostetaan tapahtuma *KeyUp* ensin A:lle ja sitten Vaihdolle.

6.14. Näppäimistötapahtumien oikea käsittely

Näppäimistötapahtumat on tärkeää käsitellä oikein, jottei kerran kulutettua tapahtumaa kuluteta vahingossa uudestaan jossain toisessa paikassa. Seuraavaksi esitellään hyvien tapojen mukainen tapahtuman kuluttaminen perusteluineen ylikirjoitetussa tapahtuman nostavassa metodissa sekä metodissa, jolle tapahtuma on tilattu. Sen jälkeen ohjeistetaan perusteluineen, miten metodeissa, joille tapahtuma on tilattu, tapahtuma käsitellään hyvien tapojen mukaisesti. Lopuksi tarkastellaan vielä erikoistilannetta ylikirjoitettuun tapahtuman nostavaan metodiin liittyen.

Ylikirjoitettu tapahtuman nostava metodi kuluttaa tapahtuman oikein siten, että metodi asettaa tapahtuman parametrin ominaisuuden *Handled* todeksi ja jättää kutsumatta kantaluokkansa toteutusta. Jos metodi kutsuisi tällöin kantaluokkansa toteutusta, lopulta suoritettaisiin luokan *Control* toteutus, joka nostaa tapahtuman riippumatta siitä, onko se on jo kulutettu. Minkään osapuolen ei pidä jakaa kulutettua tapahtumaa eteenpäin virhemahdollisuuksien vähentämiseksi.

Metodi, jolle tapahtuma on tilattu, taas kuluttaa tapahtuman oikein asettamalla sen parametrin ominaisuuden *Handled* todeksi. Jos metodi jättäisi ominaisuuden asettamatta, Windows Forms ei saisi tietoa, että kyseinen näppäinpainalluskin on tarkoitus kuluttaa ja sen jatkokäsittely voidaan lopettaa. Näppäimistötapahtumien noston jälkeisestä näppäinpainallusten käsittelyä esiteltiin tarkemmin kohdissa 6.11–6.12. Yhtä tärkeää on asettaa sama ominaisuus todeksi myös ylikirjoitetussa tapahtuman nostavassa metodissa.

Metodi, jolle tapahtuma on tilattu, aloittaa tapahtuman käsittelyn oikein siten, että ensin metodi tutkii parametristaan, onko tapahtuma kulutettu jo aiemmin. Kulutetun tapahtuman kohdalla metodi ei tee mitään. Jokaista metodia, jolle tapahtuma on tilattu, nimittäin kutsutaan tapahtuman kuluttamisesta riippumatta, mikä johtuu siitä, miten tapahtumien nostaminen toimii. Tapahtuman nostaminen laukaisee sarjan metodien kutsuja, ja tämän sarjan läpikäyntiä ei voida keskeyttää laukaisun jälkeen. Näin ollen, jos joku metodi merkitsee tapahtuman parametriin tapahtuman olevan jo kulutettu, kaikki loputkin metodit, joille tapahtuma on tilattu, suoritetaan merkinnästä huolimatta.

Jos ylikirjoitettu tapahtuman nostava metodi vastoin yleistä käytäntöä kutsuu ensin kantaluokan toteutusta ja vasta sen jälkeen itse käsittelee tapahtuman, tapahtuma voi olla kulutettu suorituksen palatessa kyseiselle metodille. Tämän takia metodin on tarkastettava, onko tapahtuma kulutettu. Kulutetun tapahtuman yhteydessä metodin ei pidä tehdä mitään.

7. WINDOWS FORMSIN SIVUUTTAMINEN

Tässä luvussa keskitytään siihen, miten Windows Forms sivuutetaan. Windows Formsin sivuuttamalla päästään eroon aiemmissa luvuissa kuvattujen toiminnallisuuksien rajoitteista työläyden kustannuksella. Kyseinen menetelmä kuvataan kohdassa 7.1 ([3, s. 845–877]). Kohdassa 7.2 esitetään aiemmissa luvuissa esille tulleiden ongelmien ratkaiseminen menetelmän avulla, ja kohdassa 7.3 pohditaan tapoja välttää työlästä menetelmää tulemalla näiden ongelmien kanssa toimeen. Kohdassa 7.4 taas tarkastellaan menetelmän soveltamista minimaalisessa laajuudessa kustannusten säästämiseksi.

Kohdassa 7.5 annetaan käyttöohje kohtien 7.3 ja 7.4 esittämille vaihtoehdoille. Seuraavaksi kohdassa 7.6 otetaan kantaa siihen, milloin menetelmää kannattaa käytännössä soveltaa. Lopuksi kohdassa 7.7 ehdotetaan lähtökohtaa, josta menetelmän soveltaminen voidaan aloittaa.

7.1. Vaihtoehto valmiille arkkitehtuurille

Aiemmissa luvuissa on käsitelty Windows Formsin tarjoamaa valmista arkkitehtuuria. Siinä on paljon suunnitteluratkaisuja, joissa on otettu hyvin huomioon räätälöidyn .NET-käyttöliittymän toteuttaminen. Siinä saattaa kuitenkin tulla vastaan tilanteita, joissa valmiin arkkitehtuurin rajoitteet estävät halutun toiminnallisuuden toteuttamisen. Valmiin arkkitehtuurin rajoitteista päästään eroon ottamalla käyttöön oma arkkitehtuuri.

Oman arkkitehtuurin käyttöön ottaminen tarkoittaa kaiken valmiin arkkitehtuurin menettämistä. Tästä johtuen oma arkkitehtuuri on suunniteltava ja toteutettava aiottuun käyttötarkoitukseen sopivassa laajuudessa jokaista yksityiskohtaa myöten. Yksityiskohtien määrä saattaa yllättää, koska niitä on yleensä voitu pitää itsestäänselvyyksinä. Oma arkkitehtuuri tuo siis vapauden työläyden kustannuksella.

Oma arkkitehtuuri rakennetaan siten, että tehdään räätälöity kontrolli, joka toimii säiliönä itse tehdyille kontrollien korvikkeille. Korvikkeella on samankaltainen vastuualue kuin kontrollilla, mutta korvike ei periydy luokasta *Control* vaan luokasta *System.Object*.

Korvikkeen ja kontrollin suurin ero on, ettei korvikkeella ole ikkunakahvaa, jolla tunnistaudutaan Windows API -rajapinnalle sen ymmärtämänä kontrollina. Tämän takia korvikkeesta käytetään nimeä ikkunaton kontrolli (windowless control, lightweight control). Toisin kuin kontrolli, ikkunaton kontrolli ei siis abstrahoi Windows API -rajapinnan ymmärtämää kontrollia, joten ikkunaton kontrolli on Windows API -rajapinnasta ja sen rajoitteista riippumaton.

Ikkunattomien kontrollien ja niiden säiliön suunnittelussa on otettava huomioon ulkoisena rajoitteena ainoastaan se, että vain säiliö pääsee piirtämään ja vastaanottamaan

käyttäjän syötettä perinteisellä tavalla eli valmiin arkkitehtuurin kautta. Tästä seuraa, että säiliön on välitettävä näihin liittyvät pyynnöt ja tiedonannot sisällään oleville ikkunattomille kontrolleille. Välittäminen tapahtuu tavalla, joka on täysin vapaasti valittavissa oman arkkitehtuurin suunnittelun yhteydessä. Säiliö siis piilottaa valmiin arkkitehtuurin palvelut tarjoamalla ne eteenpäin mahdollisesti uudessa muodossa.

Ikkunattomien kontrollien säiliön kantaluokaksi voidaan valita esimerkiksi luokka *Control*. Säiliön kantaluokaksi ei sovi luokka *ContainerControl*, koska se tarjoaa lisäpalveluita kontrolleille, jollaisia varten säiliötä ei olla tekemässä. Mikään ei kuitenkaan estä tekemästä räätälöityä kontrollia, joka toimii säiliönä sekä ikkunattomille kontrolleille että kontrolleille; tällöin ikkunattomat kontrollit piirtyvät tosin aina kontrollien taakse. Säiliön kantaluokaksi sopii myös luokka *Form*, jos halutaan, että ikkunattomia kontrolleja voidaan laittaa suoraan käyttöliittymäikkunan sisälle ilman välissä olevaa räätälöityä kontrollia.

Oma arkkitehtuuri todella tuo vapauden, koska ikkunattomille kontrolleille voidaan suunnitella esimerkiksi jokin merkitys ja esitystapa älyttömälle tilanteelle, jossa ikkunaton kontrolli on samaan aikaan monen ikkunattoman kontrollin sisällä sekä vieläpä itsensä sisällä kahteen kertaan. Vain mielikuvitus on rajana, mutta on tyypillistä huomata, että monet valmiin arkkitehtuurin tarjoamat ratkaisut ovat hyvin perusteltuja. Esimerkiksi ikkunattomien kontrollien luokkahierarkian ylimmät luokat voivat alkaa muistuttaa toiminnaltaan valmiiden kontrollien luokkahierarkian ylimpiä luokkia (kohta 3.1).

7.2. Valmiin arkkitehtuurin ongelmien ratkaiseminen

Valmis arkkitehtuuri sisältää muutamia aiemmissa luvuissa mainittuja rajoitteita, joista seuraa selviä ongelmia räätälöidyn .NET-käyttöliittymän toteuttamiseen. Seuraavaksi kerrataan nämä ongelmat ja tarkastellaan sitä, miten ne voidaan ratkaista omalla arkkitehtuurilla. Ongelmia aiheuttavat rajoitteet ovat kontrollikohtainen valmis kaksoispuskurointi (kohta 4.9), kontrollien rajoitettu kokonaismäärä (kohta 5.2) ja puutteet simuloidussa ja komponenttitason läpinäkyvyydessä (kohdat 5.11 ja 5.12).

Koska valmis kaksoispuskurointi on kontrollikohtaista, käyttäjä voi huomata kontrollien piirtyvän vuorotellen. Tämä rajoite voidaan poistaa omalla arkkitehtuurilla siten, että ikkunattomien kontrollien säiliö ottaa käyttöönsä valmiin kaksoispuskuroinnin tai toteuttaa kaksoispuskuroinnin itse koko piirtoalueelleen. Tällöin säiliö toteuttaa ikkunattomille kontrolleille kaksoispuskuroinnin, joka ei ole erillinen jokaisessa ikkunattomassa kontrollissa, ja käyttäjä ei voi mitenkään huomata, että ikkunattomien kontrollien piirtokoodit suoritetaan vuoron perään.

Koska kontrollien kokonaismäärä on rajoitettu Windowsin istuntoa ja prosessia kohti, jokaisesta mielekkäästä loogisesta kokonaisuudesta ei voi aina tehdä kontrollia. Tämä rajoite poistuu omaa arkkitehtuuria käytettäessä, koska ikkunattomien kontrollien kokonaismäärä on rajoitettu ainoastaan saatavilla olevan muistin määrän suhteen.

Simuloitu läpinäkyvyys on puutteellinen, koska se ei ota huomioon piirtojärjestystä, ja komponenttitason läpinäkyvyys taas ei toimi luotettavasti. Oma arkkitehtuuri ratkai-

see ongelman siten, että ikkunattomien kontrollien välille on mahdollista toteuttaa läpinäkyvyys oikein.

Oma arkkitehtuuri ei kuitenkaan paranna tilannetta valmiin arkkitehtuurin rajoitteista seuraavien sellaisten ongelmien kanssa, jotka ratkeavat kontrollien kanssa samoin kuin ikkunattomien kontrollienkin. Esimerkki tällaisesta ongelmasta on käyttöliittymäikkunan sovittaminen taustakuvaansa (kohta 5.10). Sovittaminen on ongelmallista, koska toinen käytettävissä olevista tavoista on työläs ja toinen taas ei toimi kaikissa tilanteissa. Ongelma voidaan ratkaista yleiskäyttöisesti esimerkiksi suunnittelemalla mekanismi, jossa käyttöliittymäikkunan taustakuvan läpinäkyvän värin tai alfa-kanavan perusteella asetetaan automaattisesti käyttöliittymäikkunan ominaisuus *Region*.

7.3. Valmiin arkkitehtuurin ongelmien hyväksyminen

Kuten kohdassa 3.1 todettiin, valmiiden kontrollien luokkahierarkian ylimmistä luokista periytetyn räätälöidyn kontrollin vastuulle jää käyttäytymisensä ja ulkoasunsa toteuttaminen suurilta osin alusta asti, mikä on työlästä. Tähän verrattuna oman arkkitehtuurin edellyttämä työmäärä on moninkertainen, koska ikkunattomien kontrollien säiliö ja ikkunattomat kontrollit eivät saa omaan arkkitehtuuriin liittyviä asioita valmiina: kaikki on erikseen suunniteltava ja toteutettava. Esimerkkejä näistä asioista on esitelty aiemmissa luvuissa, joissa on käsitelty valmiiseen arkkitehtuuriin liittyviä palveluita ja ratkaisuja.

Koska oman arkkitehtuurin suunnittelu on työlästä, kannattaa pyrkiä käyttämään valmista arkkitehtuuria hyväksyen sen ongelmat ja tullen niiden kanssa toimeen. Oman arkkitehtuurin tarvetta voidaan välttää tiedostamalla valmiin arkkitehtuurin rajoitteet ja suunnittelemalla räätälöity .NET-käyttöliittymä niiden ehdoilla. Tällaisesta suunnittelusta annetaan seuraavaksi esimerkit valmiiseen kaksoispuskurointiin, kontrollien kokonaismäärään ja läpinäkyvyyteen liittyen (kohta 7.2).

Kontrollikohtaiseksi rajoittuneen valmiin kaksoispuskuroinnin kanssa tullaan toimeen tyytymällä .NET-käyttöliittymään, jossa kontrollit piirtyvät vuorotellen havaittavasti. Käyttäjän on kuitenkin oltava tarkkaavainen havaitakseen kyseisen esteettisen epäkohdan. Jos epäkohdan havaitseminen ei edellytä tarkkaavaisuutta, kannattaa panostaa välkkymättömään .NET-käyttöliittymään esimerkiksi kohdissa 4.7, 4.9 ja 4.10 esitellyillä menetelmillä. Ikkunattomat kontrollit ratkaisisivat ongelman varmasti, mutta työläyden kustannuksella.

Kontrollien kokonaismäärää koskevan rajoitteen kanssa voidaan tulla toimeen tyytymällä käyttämään kontrolleja säästäviä ratkaisuja. Sellaisia ovat esimerkiksi näkymän vaihdon yhteydessä tapahtuva vanhan näkymän kontrollien hävittäminen ja uuden näkymän kontrollien luominen [15] sekä kontrollien uudelleenkäyttäminen eri näkymissä [16]. Kontrolleja voidaan käyttää uudelleen myös esimerkiksi taulukossa, jonka solut eivät ole kontrolleja ja jonka solujen tekstiä käyttäjän on tarkoitus voida muokata. Tällöin riittää käyttää vain yhtä tekstikenttää kuvaavaa kontrollia, joka näytetään vain muokkauksen yhteydessä muokattavan solun kohdalla [16].

Simuloidun ja komponenttitason läpinäkyvyyden puutteisiin liittyvän rajoitteen kanssa taas tullaan toimeen tyytymällä välttämään päällekkäisiä ja samalla läpinäkyviä kontrolleja jo suunnitteluvaiheessa. Sellaisia kontrolleja vastaavat ikkunattomat kontrollit eivät välttämättä kuitenkaan toisi lisäarvoa yhtä paljon kuin edellyttäisivät työtä. Kannattaa yleensäkin harkita ensin, onko tavoiteltu toiminnallisuus varmasti sen edellyttämän työn arvoista.

7.4. Esimerkki minimaalisesta toteutuksesta

Jos oman arkkitehtuurin suunnittelua ja sen mukanaan tuomaa työläyttä ei voida välttää, kannattaa minimoida oman arkkitehtuurin tarjoamien palveluiden laajuus käyttötarkoitukseensa sopivaksi. Käyttötarkoitus voi olla hyvinkin suppea. .NET-käyttöliittymässä voi olla käytössä myös useita erilaisia omia arkkitehtuureja, jotka ratkaisevat joitakin harvinaisia erityispiirteitä sisältäviä ongelmia. Sen sijaan valmiin arkkitehtuurin kaikkien palveluiden suunnittelun ja toteuttamisen pitäisi olla hyvin perusteltu.

Hyvin suppea oma arkkitehtuuri voidaan saada aikaan muutamassa työpäivässä. Esimerkiksi erään oman arkkitehtuurin suunnittelu ja toteuttaminen onnistui kahdessa työpäivässä, kun tiedettiin, mitä tehtiin. Siitä huolimatta esille tuli yllättäviä asioita, jotka oli otettava huomioon minimaalisissa palveluissa. Ne suunniteltiin ottamalla mallia valmiista arkkitehtuurista sekä toteutettiin ikkunattomien kontrollien säiliöön ja niiden kantaluokkaan, joka vastasi luokkaa *Control*.

Säiliöön toteutetut palvelut olivat sisällä olevien ikkunattomien kontrollien hallinnointi sekä niiden kokoelman hakeminen ja muuttaminen. Kokoelmaa varten tehtiin luokkaa *Control.ControlCollection* vastaava luokka, mutta suppeammalla rajapinnalla. Lisäksi säiliö käytti valmiin arkkitehtuurin tarjoamista palveluista kontrollin piirtoalueen osan merkitsemistä epäkelvoksi sekä piirtämistä ja hävittämistä.

Ikkunattomien kontrollien kantaluokkaan toteutetut palvelut olivat sisältävän säiliön hakeminen, siirtäminen uuteen säiliöön, sijainnin ja koon hakeminen ja muuttaminen, ikkunattoman kontrollin piirtoalueen osan merkitseminen epäkelvoksi, piirtäminen, piilottaminen ja hävittäminen. Näitä varten oli lisäksi toteutettava koordinaattimuunnokset säiliön ja ikkunattomien kontrollien välille.

Oman arkkitehtuurin ei pitänyt tarjota monia palveluita. Ikkunattomille kontrolleille ei suunniteltu esimerkiksi tukea toistensa sisältämiselle eikä näppäimistön ja hiiren syötteen käsittelylle kohdistuksesta puhumattakaan. Palveluita oli vähän, koska oma arkkitehtuuri ratkaisi pelkkään piirtämiseen liittyvän ongelman, joka seurasi simuloidusta läpinäkyvyydestä ja kontrollien päällekkäisyydestä. Ratkaisuna simuloidun läpinäkyvyyden käyttöä jatkettiin, mutta kontrollien päällekkäisyys poistettiin muuttamalla kontrolleja ikkunattomiksi kontrolleiksi, mikä oli helpoin ratkaisu kyseisessä tilanteessa.

Simuloitua läpinäkyvyyttä ryhdyttiin alunperin käyttämään, koska ei ollut ilmeistä, että simuloidusti läpinäkyvän kontrollin piirtoaluetta on joskus laajennettava ulottumaan viereisen kontrollin päälle. Tämä on hyvä esimerkki tilanteesta, jossa räätälöity .NET-käyttöliittymä suunnitellaan valmiin arkkitehtuurin rajoitteiden ehdoilla tiedostaen ne ja

tullen niiden kanssa toimeen helpoimmalla mahdollisella tavalla. Simuloidun läpinäkyvyyden ottaminen käyttöön vei vain vähän aikaa, ja vasta sitten, kun simuloidusta läpinäkyvyydestä seurasi ongelmia ennakoimattoman muutoksen johdosta, läpinäkyvyyteen panostettiin enemmän.

7.5. Kustannustehokas toiminta

Kohdassa 7.3 esiteltiin vaihtoehto välttää oman arkkitehtuurin tarvetta, ja kohdassa 7.4 esiteltiin vaihtoehto minimoida välttämättömän oman arkkitehtuurin palvelut. Seuraavaksi näiden vaihtoehtojen käyttöön esitetään kustannustehokkuutta tavoitteleva toimintamalli, jossa on otettu huomioon se, että oma arkkitehtuuri tuo vapauden työläyden kustannuksella.

Toimintamallissa viitataan osapuoliin asiakas ja suunnittelija, jotka voivat olla samakin henkilö. Asiakas on osapuoli, joka esittää räätälöidyn .NET-käyttöliittymän vaatimukset jonkun sovellusalan näkökulmasta sekä vastaa kustannuksista, ja suunnittelija on osapuoli, joka taas arvioi suunnittelun ja toteuttamisen edellyttämän ajan.

Ennen toteutustyön alkamista suunnittelija suunnittelee räätälöidyn .NET-käyttöliittymän vaatimusten perusteella. Tätä varten suunnittelija aloittaa työt tutustumalla valmiiseen arkkitehtuuriin lukemalla sen dokumentaatiota ja mahdollisesti aiheeseen liittyvän kirjankin. Suunnittelija tekee myös pienimuotoisia tekniikkakokeiluja tutustumisen apuna varsinkin asioista, jotka ovat suunnittelijalle uusia tai vaikuttavat ilmeisiltä haasteilta. On suunnittelijan tehtävä onnistua perustelemaan asiakkaalle, että projektin alussa tehty perehtyminen valmiiseen arkkitehtuuriin säästää projektin kokonaisaikaa.

Suunnittelija perehtyy sovellusalaankin, minkä jälkeen hän aloittaa räätälöidyn .NET-käyttöliittymän suunnittelun. Samalla suunnittelija tunnistaa ne asiat, jotka valmis arkkitehtuuri tekee kohtuuttoman hankalaksi suunnitella tai toteuttaa. Jos suunnittelija ei keksi näihin asioihin kustannustehokasta ratkaisua suppean oman arkkitehtuurin harkitsemisesta huolimatta, suunnittelija kyseenalaistaa työläät vaatimukset vieden ne asiakkaan eteen.

Suunnittelija ja asiakas pohtivat yhdessä työläiden vaatimusten hyödyllisyyttä käyttäjälle verraten sitä suunnittelijan arvioon vastaavasta lisätyöstä. Jos vaatimukset osoittautuvat huonosti perustelluiksi, ne sivuutetaan. Muutoin suunnittelu jatkuu siten, että suunnittelija ottaa valmiista arkkitehtuurista kaiken mahdollisen irti, mutta alkaa käyttää myös yhtä tai useampaa omaa arkkitehtuuria pakon sanelemana kustannustehokkuuteen tähdäten. Näin toimimalla suunnittelija välttää tarpeetonta työläyttä saaden kuitenkin riittävän vapauden käyttöönsä.

Toimintamallin kustannustehokkuus perustuu siis siihen, että suunnittelija suunnittelee valmiin arkkitehtuurin ehdoilla niin kauan kuin kustannustehokkaasti on mahdollista. Tästä huolimatta suunnittelija yrittää myös ottaa huomioon tulevaisuuden ilmeisimmät muutostarpeet ja lisävaatimukset. Lisäksi suunnittelija esittää asiakkaalle erilaisia vaihtoehtoja työmääräarvioineen ja vaikutuksineen räätälöidyn .NET-käyttöliittymän laajennettavuuteen.

Ikkunattomien kontrollien välttäminen viimeiseen asti ja minimaalisten palveluiden suunnittelu ei ole kuitenkaan kaikissa tilanteissa viisasta. Jos on selvästi näköpiirissä, että yhtenäiselle, suurelle ja yleiskäyttöiselle omalle arkkitehtuurille on tarvetta myöhemmissäkin projekteissa ja niitä on varmasti lukuisia, kannattaa harkita laajemman projektin käynnistämistä. Päätöksenteon apuna voidaan käyttää arviota mahdollisesta säästöstä verrattuna vaihtoehtoon edetä tarvekohtaisesti sekä mielikuvaa siitä, kuinka hyvin yleiskäyttöisen oman arkkitehtuurin vaatimukset ovat oikeasti tiedossa.

7.6. Kontrolli vai ikkunaton kontrolli

Kohdan 7.5 mukaan ikkunattomia kontrolleja kannattaa käyttää silloin, kun käyttö on perusteltua kustannustehokkuudella. Tässä kohdassa otetaan konkreettisemmin kantaa siihen, milloin on oikea hetki alkaa käyttää ikkunattomia kontrolleja. Tätä varten ensin täsmennetään ikkunattoman kontrollin määrittelyä. Sen jälkeen esitellään ongelma räätälöidyn .NET-käyttöliittymän jakamisesta mielekkäisiin loogisiin kokonaisuuksiin, ja lopuksi esitetään ohje ikkunattomien kontrollien valintaan.

Ikkunattomalla kontrollilla on siis samankaltainen vastuualue kuin kontrollillakin on. Sen vastuualueeseen kuuluu esimerkiksi itsensä piirtäminen ja näppäimistön syötteen käsittely. Tämän takia triviaalit tietorakenteet, jotka ovat kaukana kontrollin vastuualueesta, eivät kuulu ikkunattomien kontrollien määritelmän tarkoittamien kokonaisuuksien piiriin. Esimerkiksi totuusarvoja sisältävä taulukko ei sisällä ikkunattomia kontrolleja, vaikka jokaisen totuusarvon kohdalla piirretäänkin neliö tai ympyrä. Tällöin taulukko edustaa hienojakoisempaa arkkitehtuuria kuin mitä tarkoitetaan ikkunattoman kontrollin muodostamalla kokonaisuudella.

Ikkunaton kontrolli ei siis ole mikä tahansa tietorakenne, joka vastaa suoraan ulkoasun jotain elementtiä. Ulkoasun elementeillä tarkoitettiin käyttöliittymän osan ulkoasun osia, joihin ulkoasu voidaan mielekkäästi jakaa (kohta 4.2). Tällaisten hienojakoista arkkitehtuuria edustavien tietorakenteiden käyttäminen ei siis ole huono ajatus sen takia, etteivät ne olekaan ikkunattomia kontrolleja; kyseessä olevat tietorakenteet ainoastaan erotetaan ikkunattomista kontrolleista niiden määritelmässä.

Ikkunattoman kontrollin jokainen aliluokka kuitenkin on aina ikkunaton kontrolli periytymisen yleisten periaatteiden mukaisesti. Näin on riippumatta siitä, millainen vastuualuejako on voimassa kantaluokan ja aliluokan välillä. Ollakseen ikkunaton kontrolli, aliluokan esimerkiksi ei ole pakko toteuttaa piirtokoodia, vaan aliluokan riittää tarjota kantaluokalleen tietoa itsensä piirtämisestä muodossa, jota kantaluokka tukee. Esimerkiksi sellainen aliluokka on ikkunaton kontrolli, joka lisää kantaluokkanaan olevan ikkunattoman kontrollin tietosisältöön yhden totuusarvon, joka määrittää, piirtääkö kantaluokka neliön vai ympyrän.

Räätälöity .NET-käyttöliittymä on jaettava suunnitteluvaiheessa hierarkkisiin mielekkäisiin loogisiin kokonaisuuksiin, jotka voidaan edelleen suunnitella ja toteuttaa. Kokonaisuuden voi tehdä mielekkääksi esimerkiksi keskinäinen tiivis yhteys johonkin samaan tietoon tai toiminnallisuuteen. Esimerkiksi samantapaisina toistuvat ulkoasun ele-

menttikokoelmat, kuten painikkeet, ovat selviä mielekkäitä loogisia kokonaisuuksia. Painikkeet voivat edelleen sisältää vastaavia kokonaisuuksia, jos painikkeen ulkoasussa on monimutkaisia elementtejä.

Mielekkäiden loogisten kokonaisuuksien hierarkia voi jatkua mielivaltaisen syvälle riippumatta siitä, ovatko jaoteltavat asiat enää näkyvissä käyttäjälle. Painikkeen ulkoasun elementti voi esimerkiksi riippua neljästä asiasta, jotka on monimutkaista päätellä, jolloin neljä asiaa edelleen muodostavat omat mielekkäät loogiset kokonaisuutensa. Esimerkkinä tällaisesta tilanteesta on hätäseis-painike, jonka ulkoasun yksi elementti näyttää reaaliaikaisen koostetiedon neljän erillisen, useista laitteista koostuvan järjestelmän toimintatilasta.

Kun räätälöity .NET-käyttöliittymä on paloiteltu suunniteltaviin ja toteutettaviin osiin, on päätettävä, mistä osista tulee kontrolleja ja mitkä jäävät hienojakoista arkkitehtuuria edustaviksi tietorakenteiksi. Liiallinen kontrollien käyttö saattaa johtaa tarpeettomaan monimutkaisuuteen, johon toisaalta päädytään toisenkin ääripään kanssa.

Jaottelu kontrolleihin ei voi kuitenkaan tapahtua yksinkertaisuuden ehdoilla, koska kontrollien kokonaismäärä on rajoitettu (kohta 5.2). Esimerkiksi on mahdotonta tehdä tekstinkäsittelyohjelma, jonka jokainen merkki on oma kontrollinsa, vaikka tällainen ratkaisu olisikin arkkitehtuurin kannalta kaunis (ajatus peräisin lähteestä [10] nimimerkiltä Adrian). Valmis arkkitehtuuri on suunniteltu huonosti sisältäessään tällaisen rajoitteen; sen kiertämisestä ikkunattomien kontrollien avulla seuraa ylimääräistä vaivaa.

Kontrollien valinnassa on siis ensisijaisesti kiinnitettävä huomiota siihen, pakottaako niiden rajoitettu kokonaismäärä selvästi käyttämään ikkunattomia kontrolleja. Kontrollien valinnassa on myös pidettävä mielessä, että ikkunattoman kontrollin sisälle voi laittaa ainoastaan muita ikkunattomia kontrolleja tai hienojakoista arkkitehtuuria edustavia tietorakenteita. Tästä seuraten, kannattaa harkita ikkunattomien kontrollien käyttämistä varsinkin niissä mielekkäissä kokonaisuuksissa, jotka tuhlaisivat kontrolleja eniten. On kuitenkin pyrittävä minimoimaan oman arkkitehtuurin palvelujen laajuus esimerkiksi välttämällä tarvetta käsitellä syötettä ikkunattomissa kontrolleissa.

Ikkunattomien kontrollien tarve voidaan sivuuttaa siten, että ne kontrollit, jotka eivät mahdu kontrolli-budjettiin, korvataan muiden kontrollien sisälle tulevilla hienojakoista arkkitehtuuria edustavilla tietorakenteilla. Jos kuitenkin niiden tilalla olisi ollut luontevaa käyttää ikkunattomia kontrolleja, niiden välttämisestä seuraa monimutkaisuutta kyseisissä tietorakenteissa tai niiden käsittelyssä. Tällöin ikkunattomat kontrollit olisivat voineet säästää aikaa.

Ikkunattomiin kontrolleihin voidaan päätyä perustellusti myös muista syistä. Jos valmis arkkitehtuuri sisältää rajoitteita, joiden kiertämiseen on käytettävä ikkunattomia kontrolleja, ja tavoiteltu toiminnallisuus on työläyden arvoista, ikkunattomat kontrollit ovat selvä valinta. Esimerkiksi simuloidun ja komponenttitason läpinäkyvyyden puutteet voivat olla tällaisia syitä.

Valintaohje kontrollin ja ikkunattoman kontrollin välillä voidaan tiivistää seuraavaksi luetteloksi. Valintaohjeen soveltaminen edellyttää sekä valmiin arkkitehtuurin rajoitteiden tuntemista että vaatimusten perusteiden ymmärtämistä.

- Valitse ensisijaisesti aina kontrolli, mutta, jos valinnan seurauksena jonkin perustellun vaatimuksen ottaminen huomioon vaikeutuu huomattavasti valmiin arkkitehtuurin rajoitteen takia, valitse ikkunaton kontrolli.
- Pidä myös mielessä valintaa tehdessä, että ikkunattomat kontrollit voidaan suunnitella ja toteuttaa äärimmäisen kevyestikin jotain rajattua käyttötarkoitusta varten, miten tehtiin kohdan 7.4 esimerkissä. Ikkunattomat kontrollit voivat hyvin olla niinkin yksinkertaisia, että ainoastaan seuraavat jotain tietomallia piirtäen sen mukaisen kuvan sijaintiin, johon ikkunaton kontrolli on laitettu.

7.7. Esimerkki omasta arkkitehtuurista

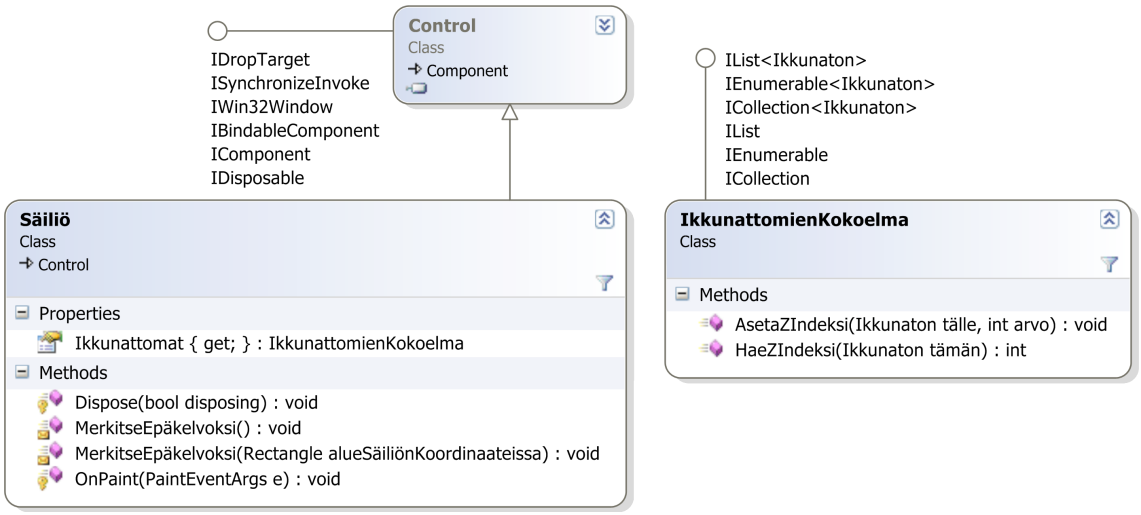
Oma arkkitehtuuri riippuu paljon käyttötarkoituksesta, minkä takia ainoaa oikeaa ratkaisua ei ole olemassa. Seuraavaksi esitetään suppea ehdotus oman arkkitehtuurin pohjasta herättämään ajatuksia kaikesta siitä, mitä omaa arkkitehtuuria varten on suunniteltava ja toteutettava. Ehdotuksessa käytetään samantapaisia ratkaisuja kuin valmiissa arkkitehtuurissa on käytetty, joten arkkitehtuureista voidaan tunnistaa vastaavia, eri tavoin nimettyjä luokkia ja niiden jäseniä sekä parametreja.

Seuraavaksi esitellään kuvia, joissa on oman arkkitehtuurin keskeisimmät luokkakaaviot. Niissä käytetään jäsenten näkyvyysmääreiden kuvaamiseen ohjelmankehitysympäristöstä Visual Studio 2005 tuttuja symboleja. Lukko tarkoittaa yksityistä (private), avain suojattua (protected) ja kirjekuori sisäistä (internal). Muiden jäsenten näkyvyysmääre on julkinen (public). Abstraktit jäsenet ovat kursiivilla. Luokkakaavioissa on näkyvillä ainoastaan aiheen kannalta mielenkiintoiset jäsenet.

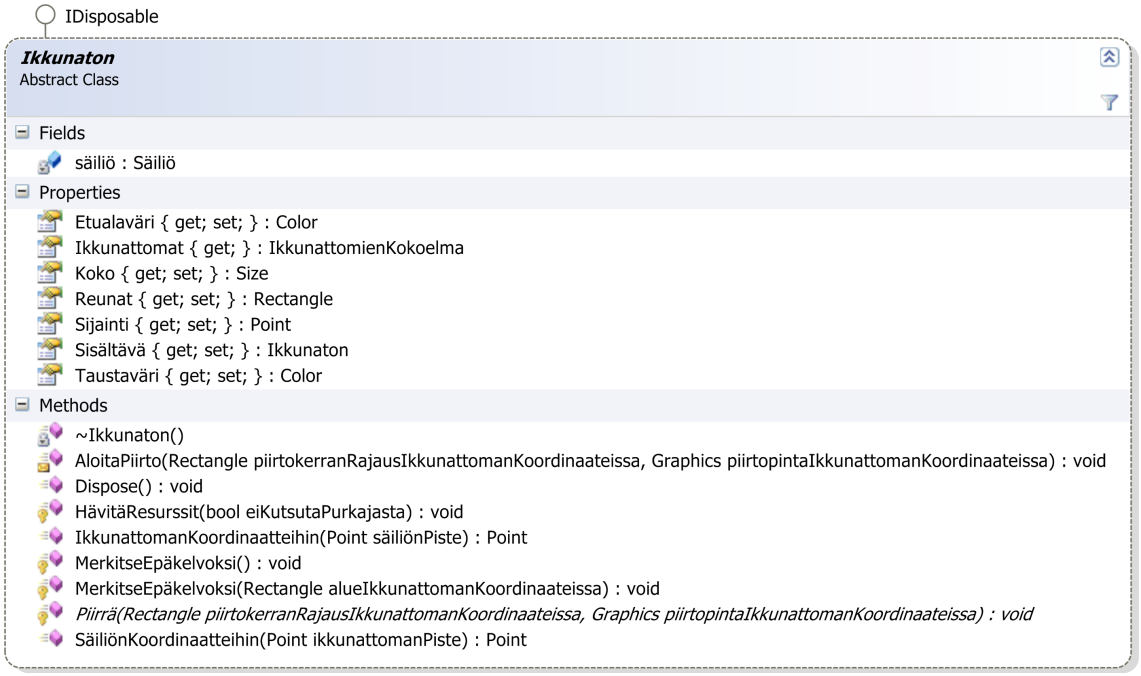
Oma arkkitehtuuri koostuu luokista *Säiliö*, *IkkunattomienKokoelma* ja *Ikkunaton*, joiden luokkakaaviot ovat kuvissa 7.1 ja 7.2. *Säiliö* on räätälöity kontrolli, joka pääsee käyttämään valmiin arkkitehtuurin palveluita. *IkkunattomienKokoelma* on kokoelma ikkunattomia kontrolleja, ja *Ikkunaton* on taas ikkunattomien kontrollien kantaluokka. Luokan *Säiliö* vastuulla on tarjota valmiin arkkitehtuurin palvelut luokalle *Ikkunaton*. Sen vastuulla on taas tarjota aliluokilleen oman arkkitehtuurin palvelut.

Luokan *IkkunattomienKokoelma* vastuulla on ikkunattomien kontrollien sisältyvyyksien sekä ikkunattomien kontrollien z-järjestyksen hallinnointi. Sekä luokalla *Säiliö* että luokalla *Ikkunaton* on ominaisuus *Ikkunattomat*, jossa on kokoelma sisällä olevia ikkunattomia kontrolleja. Sisältyvyys on linkitetty toiseenkin suuntaan, koska luokalla *Ikkunaton* on ominaisuus *Sisältävä*, jossa on sisältävä ikkunaton kontrolli. Jos ikkunaton kontrolli on suoraan luokan *Säiliö* sisällä, kyseisessä ominaisuudessa on arvo null. Tällainen ikkunaton kontrolli on ainoa, joka käyttää kenttäänsä *säiliö*.

Sekä *Säiliö* että *Ikkunaton* toteuttavat rajapinnan *IDisposable* .NET-rajapinnan käyttämän suunnittelumallin mukaisesti. Tämän takia *Säiliö* kutsuu parametrillisesta metodistaan *Dispose* jokaisen sisällään olevan ikkunattoman kontrollin saman nimistä parametritonta metodia, jotka edelleen kutsuvat samaa parametritonta metodia sisällään oleville ikkunattomille kontrolleille.



Kuva 7.1. *Oman arkkitehtuurin luokat Säiliö ja IkkunattomienKokoelma*



Kuva 7.2. *Oman arkkitehtuurin luokka Ikkunaton*

Kantaluokan *Ikkunaton* metodi *Dispose* kutsuu aliluokkiaan varten ylikirjoitettavaa metodia *HävitäResurssit* ohjeistaen parametrilla hävittämään sekä hallinnoidun että hallinnoimattoman koodin (unmanaged code) resurssit. Kantaluokan *Ikkunaton* purkaja kutsuu myös samaa metodia vapauttaakseen vain viimeksi mainitut resurssit. Metodin *HävitäResurssit* ylikirjoittavan aliluokan on tarkoitus kutsua lopuksi kantaluokan toteutusta, ja kyseisen metodin toteutus kantaluokassa *Ikkunaton* vastaa jokaisen sisällä olevan ikkunattoman kontrollin metodin *Dispose* kutsumisesta.

Luokka *Säiliö* kutsuu metodistaan *OnPaint* jokaisen sisällä olevan ikkunattoman kontrollin metodia *AloitaPiirto*, joka puolestaan kutsuu saman ikkunattoman kontrollin

metodia *Piirrä*. Tämän jälkeen rekursiivinen metodi *AloitaPiirto* kutsuu edelleen jokaisen sisällä olevan ikkunattoman kontrollin samaa metodia.

Kantaluokka *Ikkunaton* tarjoaa aliluokilleen metodin *MerkitseEpäkelvoksi*, jonka avulla ikkunaton kontrolli voi ilmoittaa piirtoalueelleen viimeksi piirretyn sisällön vanhentuneen. Kyseinen metodi kutsuu sisältävän ikkunattoman kontrollin toteutusta rekursiivisesti, kunnes ominaisuuden *Sisältävä* arvo on null. Tällöin käytetään kenttää *säiliö*, jotta päästään käsiksi luokan *Säiliö* saman nimiseen metodiin. Se puolestaan kutsuu räätälöidyn kontrollin *Säiliö* metodia *Invalidate*.

Luokalla *Ikkunaton* on ominaisuudet *Sijainti* ja *Koko* sekä ne koostava ominaisuus *Reunat*. Nämä ominaisuudet kuvaavat asetteluun liittyvää tietoa sisältävän osapuolen koordinaateissa. Kyseisten ominaisuuksien toteutuksessa on otettava huomioon arvon asettamisen yhteydessä, että ikkunattoman kontrollin koko piirtoalue on ilmoitettava vanhentuneeksi sekä vanhassa sijainnissaan että uudessa. Muutoin ikkunaton kontrolli voi näyttää kopioituvan tai jäävän alkuperäiseen asetteluunsa.

Jotta kantaluokan *Ikkunaton* aliluokkien toteuttaminen helpottuisi, kantaluokkaan on toteutettu koordinaattimuunnokset säiliön koordinaatteihin ja takaisin. Niiden lisäksi luokan *Säiliö* metodi *OnPaint* sekä luokan *Ikkunaton* metodit *AloitaPiirto* ja *MerkitseEpäkelvoksi* suorittavat koordinaattimuunnoksia metodien rekursiivisen suorituksen aikana. Tämän ansiosta esimerkiksi ikkunattoman kontrollin metodi *Piirrä* saa aina piirtopinnan, joka on siirretty ikkunattoman kontrollin omiin koordinaatteihin.

Kantaluokalla *Ikkunaton* on ominaisuudet *Etualaväri* ja *Taustaväri*, joita aliluokat voivat käyttää itsensä piirtämisessä. Valmiista arkkitehtuurista voi halutessaan ottaa mallia enemmänkin toteuttamalla vastaavia ominaisuuksia lisää. Niistä voi lisäksi halutessaan tehdä ympäröiviä (kohta 5.5). Vastaavasti luokalla *Ikkunaton* voi myös olla ketjutetut ominaisuudet *Näkyvillä* ja *Käytettävissä* (kohta 5.4).

Kuvassa 7.3 on esimerkkejä ikkunattomista kontrolleista. Ne ylikirjoittavat metodit *HävitäResurssit* ja *Piirrä* sekä käyttävät kantaluokkansa palveluita. Esimerkiksi ikkunaton kontrolli *AnaloginenKello* käyttää kantaluokkansa ominaisuutta *Etualaväri* viisarin värinä ja ominaisuutta *Taustaväri* kellotaulun taustavärinä, jos luokan *AnaloginenKello* ominaisuuden *Taustakuva* arvo on null.

Kuten räätälöityjenkin kontrollien kohdalla, jokaisen ikkunattoman kontrollin ei ole pakko sisältää piirtämisestä vastaavaa metodia, vaan riittää, että itse piirtämisestä vastaa jokin kantaluokista; aliluokka tällöin vain määrittelee piirrettävän sisällön. Esimerkiksi kuvan luokka *Kuvio* edellyttää aliluokkiensa toteuttavan ainoastaan abstraktin ominaisuuden *Reunakäyrä*, jonka avulla luokka *Kuvio* osaa huolehtia piirtämisestä. [3, s. 845–877.]

Kuva 7.3. *Ikkunattomia kontrolleja*

Koska oma arkkitehtuuri tuo mukanaan vapauden, valmiin arkkitehtuurin kopioimiseen ei ole kuitenkaan mikään pakko rajoittua. Esimerkiksi piirtomallin ei ole pakko olla kopio Windows Formsista, joka käyttää Windows API -rajapinnan piirtomallia. Se, monen muun ohella, suunniteltiin 1985-luvulla käytettävissä olevaan tietokoneen muistin määrän ehdoilla [3, s. 214; 9]. Valmiin arkkitehtuurin piirtomallissa käyttöliittymän osa piirtää aina tullessaan näkyville jonkun osan takaa [3, s. 214–215]. Piirtomalli voi hyvin olla sellainenkin, jossa käyttöliittymän osa piirtää vain muutokset piirrettävässä sisällössä, ja edellinen piirretty sisältö säilytetään muistissa [3, s. 214–215; 17].

8. YHTEENVETO

Tämän diplomityön anti on jäsennelty ja sulateltu esitys siitä, miten tehdään räätälöity .NET-käyttöliittymä. Esityksen kattamat aihealueet ovat välkkymättömyys, läpinäkyvyys, kohdistus ja näppäimistön syötteen käsittely sekä mahdollisuus päästä vaikuttamaan yksityiskohtiin Windows Formsin rajoitteet ohittaen.

Diplomityön merkitys on osaamisen syventyminen suunnittelun, toteuttamisen ja alan kirjallisuuden sekä kirjallisen esityksen muotoilun avulla. Tärkeänä merkityksenään diplomityö myös jakaa opittuja asioita helpossa muodossa aiheesta kiinnostuneelle lukijalle. Aiheeseen perehtymätön voi diplomityön avulla tutustua räätälöidyn .NET-käyttöliittymän tekemiseen liittyviin yksityiskohtiin sen sijaan, että joutuisi toistamaan sulattelutyön, jonka tulokset ovat diplomityössä.

Diplomityö antaa suosituksia konkreettisiksi toimenpiteiksi osin kokemukseen perustuen. Suositukset tuodaan seuraavaksi esille samassa järjestyksessä kuin ne esiintyvät tekstissä.

- Valitse räätälöidyn kontrollin kantaluokaksi mahdollisimman paljon sopivia palveluita valmiina tarjoava vaihtoehto. Seuraa kantaluokan tapahtumaa ylikirjoittamalla tilaamisen sijaan.
- Ota huomioon räätälöidyn .NET-käyttöliittymän välkkymättömyys tutustumalla piirtämisen optimoinnin perusideoihin jo projektin alussa.
- Suunnittele räätälöity .NET-käyttöliittymä ensisijaisesti valmiin arkkitehtuurin ehdoilla. Kiinnitä erityisesti huomiota kontrollien kokonaismäärään sekä samaan aikaan läpinäkyvien ja päällekkäisten kontrollien välttämiseen.
- Toteuta hiirellä käytettävälle kontrollille mahdollisuus näppäimistönkin käytölle kohdistusmahdollisuuden poistamisen yrittämisen sijaan. Jos kontrollia ei voi käyttää hiirellä eikä näppäimistöllä, poista kohdistusmahdollisuus estämällä kontrollin käyttö.
- Käsittele näppäimistön syöte ensisijaisesti näppäimistötapahtumilla ja hyvien tapojen mukaisesti. Jos näppäinpainallus ei johda näppäimistötapahtumaan, poista erikoisnäppäinten näppäinpainallusten käsittely. Käytä räätälöidyssä kontrollissa hyväksi valmiin arkkitehtuurin erikoisnäppäimille tarjoamia ratkaisuja.
- Valmiin arkkitehtuurin rajoitteista päästään eroon omalla arkkitehtuurilla työläyden kustannuksella. Työläyden takia vältä oman arkkitehtuurin tarvetta ja laajuutta. Muista silti arvioida laajan yleiskäyttöisen oman arkkitehtuurin mukanaan tuomat säästöt tulevissa projekteissa.

Suositukset ovat yleispäteviä, joskin harvinaisemmissa käyttötarkoituksissa voi olla perusteltua toimia toisin, jos seuraukset ovat hyvin tiedossa. Yleisesti ottaen kannattaa

pyrkiä noudattamaan samantapaisia ratkaisuja kuin valmiissa kontrolleissa on käytetty. Suosituksia voi käyttää kehyksen ja Windowsin versioista riippumatta, koska uusin versio .NET Framework 4.0 pohjautuu vanhoihin ja kehyksen jokainen uusi versio lähinnä tuo mukanaan uutta sisältöä vanhempiin verrattuna.

Diplomityössä käsitellään Windows Formsia, mutta monet esitellyt ilmiöt voivat esiintyä muissakin teknisissä ratkaisuissa. Diplomityö ei vertaa niitä keskenään, mutta eräänä vaihtoehtona on esimerkiksi Windows Presentation Foundation eli WPF, joka on Microsoftin tekemä uudempi kehys. WPF on tarkoitettu vastaavaan käyttötarkoitukseen eli räätälöidyn käyttöliittymän tekemiseen Windowsiin.

Räätälöidyn .NET-käyttöliittymän tekeminen sisältää paljon muitakin näkökulmia diplomityössä esitetyn lisäksi. Teknisistä näkökulmista esimerkiksi suunnittelun kannalta on oleellista myös muistinhallinta. Tämän diplomityön näkökulmaksi valittiin kuitenkin räätälöidyn .NET-käyttöliittymän käyttäytyminen ja ulkoasu.

Tärkeänä asiana, ennen räätälöidyn käyttöliittymän tekemisen aloittamista, kannattaa hinnoitella käyttäjän parantuva käyttökokemus eli määrittää kustannukset, jotka ovat saavutetun hyödyn arvoisia. Lisähinta on aina hyvin perusteltu, jos muutoin ei voida täyttää riittäviä käytettävyyden vaatimuksia toiminnallisuuden ja esitystavan suhteen. Toisena hyvänä perusteluna on uskomus markkinoiden kohentumiseen parannellun ulkoasun myötä.

Halvin vaihtoehto on tehdä räätälöidyn käyttöliittymän sijaan mukautettu käyttöliittymä, jolla on persoonallinen käyttäytyminen ja ulkoasu. Räätälöidyn käyttöliittymän muunneltavuus on parempi, mutta kustannukset taas ovat suuremmat. Oman arkkitehtuurin sisällään pitävällä räätälöidyllä käyttöliittymällä saadaan aikaan vieläkin enemmän, mutta suureen hintaan. Jälkimmäisten vaihtoehtojen kustannuksia voi tosin olla mahdollista pienentää kaupallisia piirtokirjastoja käyttämällä.

LÄHTEET

[1] Kuutti, K. Luento 5: Käyttöliittymät [verkkodokumentti]. Tietojenkäsittelyopin laitos. Tietojenkäsittelyopin peruskurssin kotisivut, Oulun yliopisto. [viitattu 23.1.2010]. Saatavissa: http://www.tol.oulu.fi/kurssit/tkop/tkop5_1.html.

[2] .NET Framework 2.0 Class Library Reference [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 27.1.2010]. Saatavissa: http://msdn.microsoft.com/fi-fi/library/ms229335(en-us,VS.80).aspx.

[3] MacDonald, M. Pro .NET 2.0 Windows Forms and Custom Controls in C#. 1st ed. New York 2005, Apress. 1080 pp. ISBN 1-59059-439-8.

[4] .NET Framework 2.0 Conceptual Overview [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 27.1.2010]. Saatavissa: http://msdn.microsoft.com/en-us/library/zw4w595w(VS.80).aspx.

[5] .NET Framework 2.0 Windows Forms Programming: User Input in Windows Forms [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 27.1.2010]. Saatavissa: http://msdn.microsoft.com/fi-fi/library/ms171532(en-us,VS.80).aspx.

[6] .NET Framework 2.0 Windows Forms Programming: Varieties of Custom Controls [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 28.1.2010]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms171725(VS.80).aspx.

[7] Prabhu, R. Control and UserControl [verkkoblogi]. MSDN Blogs, Microsoft. Julkaistu 23.10.2003 [viitattu 23.1.2010]. Saatavissa: http://blogs.msdn.com/rprabhu/archive/2003/10/23/56549.aspx.

[8] Win32 and COM Development: User Objects [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 18.2.2010]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms725486(VS.85).aspx.

[9] Chen, R. Windows are not cheap objects [verkkoblogi]. MSDN Blogs, Microsoft. Julkaistu 15.3.2005 [viitattu 18.2.2010]. Saatavissa: http://blogs.msdn.com/oldnewthing/archive/2005/03/15/395866.aspx.

[10] Chen, R. Why is the limit of window handles per process 10,000? [verkkoblogi]. MSDN Blogs, Microsoft. Julkaistu 18.7.2007 [viitattu 18.2.2010]. Saatavissa: http://blogs.msdn.com/oldnewthing/archive/2007/07/18/3926581.aspx.

[11] Increasing GDI and User Handle Limits in Windows [nimimerkin hamesh kirjoitus, verkkokeskustelupalsta]. Windows-Now, Belchfire Themes. Julkaistu 29.8.2007 [viitattu 18.2.2010]. Saatavissa: http://www.belchfire.net/index.php?showtopic=19167.

[12] Chen, R. How are window manager handles determined in Windows NT? [verkkoblogi]. MSDN Blogs, Microsoft. Julkaistu 17.7.2007 [viitattu 18.2.2010]. Saatavissa: http://blogs.msdn.com/oldnewthing/archive/2007/07/17/3903614.aspx.

[13] Win32 and COM Development: CreateWindowEx Function [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 27.1.2010]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms632680(VS.85).aspx.

[14] Weigelt, R. ReadOnlyRichTextBox [verkkoblogi]. ASP.NET Weblogs, Microsoft. Julkaistu 9.7.2004, päivitetty 10.7.2004 [viitattu 29.3.2010]. Saatavissa: http://weblogs.asp.net/rweigelt/archive/2004/07/10/178813.aspx.

[15] Marguerie, F. The "Error creating window handle" exception and the Desktop Heap [verkkoblogi]. ASP.NET Weblogs, Microsoft. Julkaistu 7.8.2009 [viitattu 25.2.2010]. Saatavissa: http://weblogs.asp.net/fmarguerie/archive/2009/08/07/cannot-create-window-handle-desktop-heap.aspx.

[16] What to do about “Error creating window handle” errors in a C# application? [nimimerkkien jdigital ja Rossney, R. kirjoitus, verkkokeskustelupalsta]. Questions, Stack Overflow. Julkaistu 6.2.2009, päivitetty 11.2.2009 [viitattu 25.2.2010]. Saatavissa: http://stackoverflow.com/questions/522534/what-to-do-about-error-creating-window-handle-errors-in-a-c-application.

[17] .NET Framework 3.5 Windows Presentation Foundation: Graphics Rendering Overview [verkkodokumentti]. MSDN Library, Microsoft. [viitattu 29.3.2010]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms748373.aspx.